Summa Summarum: Moessner's Theorem without Dynamic Programming

Olivier Danvy
School of Computing & Yale-NUS College, National University of Singapore, Singapore
danvy@acm.org

Seventy years on, Moessner's theorem and Moessner's process – i.e., the additive computation of integral powers – continue to fascinate. They have given rise to a variety of elegant proofs, to an implementation in hardware, to generalizations, and now even to a popular video, "The Moessner Miracle." The existence of this video, and even more its title, indicate that while the "what" of Moessner's process is understood, its "how" and even more its "why" are still elusive. And indeed all the proofs of Moessner's theorem involve more complicated concepts than both the theorem and the process.

This article identifies that Moessner's process implements an additive function with dynamic programming. A version of this implementation without dynamic programming (1) gives rise to a simpler statement of Moessner's theorem and (2) can be abstracted and then instantiated into related additive computations. The simpler statement also suggests a simpler and more efficient implementation to compute integral powers as well as simple additive functions to compute, e.g., Factorial numbers. It also reveals the source of – to quote John Conway and Richard Guy – Moessner's magic.

Keywords: Moessner's theorem; Moessner's process; streams; nested summations; primitive iteration; primitive recursion; integral powers; single, double, triple, etc. Factorial numbers; superfactorial numbers; binomial coefficients; Catalan numbers; Fibonacci numbers; Euler (zigzag, up/down) numbers, polygonal numbers

---

Executive Summary

Beneath its dynamic-programming infrastructure, the foundation of Moessner's process is nested summations. Here is Moessner's theorem without dynamic programming, where each nested sum is a streamless instance of a prefix sum and each inner upper bound is a streamless instance of the filtering-out phase in Moessner's process:

$$\forall x:\mathbb{N}, \forall n:\mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{\lfloor\frac{2\cdot i_1}{1}\rfloor} \sum_{i_3=0}^{\lfloor\frac{3\cdot i_2}{2}\rfloor} \cdots \sum_{i_n=0}^{\lfloor\frac{n\cdot i_{n-1}}{n-1}\rfloor} 1 \;=\; \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+\lfloor\frac{i_1}{1}\rfloor} \sum_{i_3=0}^{i_2+\lfloor\frac{i_2}{2}\rfloor} \cdots \sum_{i_n=0}^{i_{n-1}+\lfloor\frac{i_{n-1}}{n-1}\rfloor} 1 \;=\; (x+1)^n$$

Moessner's magic [10] lies in the dependencies of the indices in these summations.

Here are some new corollaries: $\forall x:\mathbb{N}, \forall n:\mathbb{N}$,

$$\text{Binomial coefficients: } \textstyle\sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{i_2} \cdots \sum_{i_n=0}^{i_{n-1}} 1 = \binom{x+n}{n}$$

$$\text{Catalan numbers: } \textstyle\sum_{i_1=0}^{0} \sum_{i_2=0}^{i_1+1} \sum_{i_3=0}^{i_2+1} \cdots \sum_{i_n=0}^{i_{n-1}+1} 1 = \frac{\binom{2\cdot n}{n}}{n+1} = C_n$$

$$\text{Fibonacci numbers: } \textstyle\sum_{i_1=0}^{0} \sum_{i_2=0}^{1-i_1} \sum_{i_3=0}^{1-i_2} \cdots \sum_{i_n=0}^{1-i_{n-1}} 1 = F_{n+1}$$

$$\text{Euler (zigzag) numbers: } \textstyle\sum_{i_1=0}^{0} \sum_{i_2=0}^{n-1-i_1} \cdots \sum_{i_{n-1}=0}^{1-i_{n-2}} \sum_{i_n=0}^{0-i_{n-1}} 1 = E_n$$

Generalizing 0 to x in the second corollary gives rise to the $x+1$st convolution of Catalan numbers.

Bieniusa, Degen, Wehr (Eds.). A Second Soul:
Celebrating the Many Languages of Programming
EPTCS ??, 2024, pp. 57–92, doi:10.4204/EPTCS.??.5

That said, for several of the applications of Moessner's theorem in the literature – e.g., factorial numbers but also integral powers – Moessner's magic (i.e., the upper bound of an inner sum depending on the index of an outer sum) is not needed. In each of the following summations, the index does not occur in the body of this summation:

$$\text{Factorial numbers:} \quad \forall n : \mathbb{N}, \sum_{i_1=0}^{1} \sum_{i_2=0}^{2} \sum_{i_3=0}^{3} \cdots \sum_{i_n=0}^{n} 1 \ = \ (n+1)!$$

$$\text{Integral powers:} \ \forall x, n : \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{x} \sum_{i_3=0}^{x} \cdots \sum_{i_n=0}^{x} 1 \ = \ (x+1)^n$$

This observation sheds a new light on Paasche's slide rule [47] since the product of a sequence of factors can be expressed as an iterated summation:

$$\forall f : \mathbb{N} \to \mathbb{N}, \forall n : \mathbb{N}, \prod_{i=0}^{n} f(i) \ = \ \sum_{i=1}^{f(0)} \sum_{i=1}^{f(1)} \sum_{i=1}^{f(2)} \cdots \sum_{i=1}^{f(n)} 1$$

In practice, the nested summations induce many duplicated computations. Dynamic programming makes it possible to avoid these duplications, giving rise, e.g., to Pascal's triangle for binomial coefficients and more generally to additive processes in the style of Moessner that are ready to be implemented in hardware [55].

---

1 Introduction

Whereas Eratosthenes's sieve constructs a stream of successive prime numbers $[2, 3, 5, \cdots[$ given the stream of positive natural numbers $[1, 2, 3, \cdots[$, Moessner's process constructs a stream of integral powers $[1^n, 2^n, 3^n, \cdots[$ given the stream $[1, 0, 0, \cdots[$ and a natural number n. Counting down from n to 0,

(n) it filters out (strikes out / skips / drops / elides / omits) each $n+2$nd element from this given stream and then constructs the corresponding stream of prefix sums (given a stream $[x_0, x_1, x_2, \cdots[$, the corresponding stream of prefix sums is $[\sum_{i=0}^{0} x_i, \sum_{i=0}^{1} x_i, \sum_{i=0}^{2} x_i, \cdots[$, i.e., $[x_0, x_0+x_1, x_0+x_1+x_2, \cdots[$);

...

(i) it filters out each $i+2$nd element from the resulting stream and then constructs the corresponding stream of prefix sums;

...

(0) it filters out each second element from the resulting stream and then constructs the corresponding stream of prefix sums.

Moessner's theorem [35] states that the resulting stream is $[1^n, 2^n, 3^n, \cdots[$, which is immediately visible for small values of n:

- When $n = 0$, the resulting stream of prefix sums is $[1, 1+0, (1+0)+0, ((1+0)+0)+0, \cdots[$, i.e., $[1, 1, 1, 1, \cdots[$, or again $[1^0, 2^0, 3^0, 4^0, \cdots[$.
- When $n = 1$, the first stream of prefix sums is the stream of 1's and the resulting stream of prefix sums is $[1, 1+1, (1+1)+1, ((1+1)+1)+1, \cdots[$, i.e., $[1, 2, 3, 4, \cdots[$, or again $[1^1, 2^1, 3^1, 4^1, \cdots[$.
- When $n = 2$, the first stream of prefix sums is the stream of 1's and the second is the stream of positive natural numbers. Filtering out its second elements constructs the stream of odd natural numbers, and the resulting stream of prefix sums is $[1, 1+3, (1+3)+5, ((1+3)+5)+7, \cdots[$, i.e., $[1, 4, 9, 16, 25, \cdots[$, or again $[1^2, 2^2, 3^2, 4^2, 5^2, \cdots[$. Indeed, as is known since the Pythagoreans, the $x+1$st positive square integer is the sum of the first $x+1$st odd natural numbers:

$$\forall x : \mathbb{N}, \sum_{i=0}^{x} (2 \cdot i + 1) = (x+1)^2$$

since $2 \cdot i + 1 = (i+1)^2 - i^2$ because of the binomial expansion into a sum of monomials with binomial coefficients $(i+1)^2 = \binom{2}{2} \cdot i^2 \cdot 1^0 + \binom{2}{1} \cdot i^1 \cdot 1^1 + \binom{2}{0} \cdot i^0 \cdot 1^2 = i^2 + 2 \cdot i + 1$ and using the telescoping sum $\sum_{i=0}^{x} (i+1)^2 - i^2 = (1^2 - 0^2) + (2^2 - 1^2) + (3^2 - 2^2) + \cdots + (x^2 - (x-1)^2) + ((x+1)^2 - x^2) = -0^2 + (x+1)^2$.

Beyond 2, things become more involved, but the result is still remarkable because not a single multiplication is performed to obtain powers. Moessner's process has therefore been implemented in hardware for signal-processing purposes [55].

Over the years, a number of proofs for Moessner's theorem have been put forward: first, induction proofs by mathematicians in the 20th century (Long, Paasche, Perron, Salié, Slater, van Yzeren, and then Ross, as reviewed in Section 2.1) and then, both induction proofs and coinduction proofs by theoretical computer scientists in the 21st century (Bickford, Hinze, Kozen, Krebbers, Niqui, Parlant, Rutten, and Silva, as reviewed in Section 2.2). Nowadays, Moessner's process is a showcase for demonstrating stream calculi, mechanizing coinduction proofs, discovering patterns, and formulating generalizations. (Watch the video [51].)

The starting point here is the observation that Moessner's iterative process of computing successive streams fits the pattern of dynamic programming. The goal of this article is to unveil the essence of Moessner's theorem by removing this dynamic-programming infrastructure.

1.1 Roadmap

Section 2 illustrates Moessner's process and reviews previous work about Moessner's theorem. Section 3 presents a version of Moessner's process that does not use dynamic programming. Section 4 presents a version of the left inverse of Moessner's process that also does not use dynamic programming. Section 5 builds on Sections 3 and 4 and presents the essence of Moessner's theorem. Correspondingly, Section 6 implements the essence of Moessner's process. Section 7 parameterizes this implementation and illustrates it with several instantiations that reveal Moessner's magic as primitive recursion. Section 8 goes back to dynamic programming, using lists instead of streams. Section 9 reviews previous work about nested sums. Appendix C presents a new characterization of polygonal numbers in memory of Moessner.

1.2 Prerequisites and Notations

An elementary grasp of functional programming (e.g., the λ notation for functions [7] and the `lambda` notation for procedures in Scheme [17]) and of discrete mathematics [20,53] is expected from the reader. Also, we use Peano numbers, i.e., natural numbers that start with 0.

1.2.1 Functional programming

The programming language of discourse here is Scheme, a block-structured and lexically scoped dialect of Lisp with first-class procedures and proper tail recursion. (So programs are fully parenthesized expressions that use the Polish prefix notation and iteration is achieved with tail-recursive procedures.)

A list is an inductive data structure that is constructed with nil (the empty list, written `'()` in Scheme) in the base case, and with cons (implemented by the Scheme procedure `cons`) in the induction step. For example, the Scheme procedure `iota`, given a non-negative integer n, constructs the list of n payloads $0, 1, 2, \ldots, (n-1)$. So evaluating `(iota 3)` gives rise to evaluating `(cons 0 (cons 1 (cons 2 '())))` and yields the list `(0 1 2)`.

Likewise, the Scheme procedure `map`, given a procedure and a list, applies this procedure to each payload in the list and constructs the list of the results, in the same order. So for example, evaluating

```
(map (lambda (n) (+ n 1)) (iota 3))
```

yields `(1 2 3)` since `(lambda (n) (+ n 1))` implements the successor function, which is applied to each of the payloads in `(0 1 2)` here.

A stream is a coinductive data structure (intuitively: an unbounded list [18]) that is constructed on demand. Given a function $f : \mathbb{N} \to \mathbb{N}$, the stream $[f\,0, f\,1, f\,2, \cdots[$ is a lazy representation of f's function graph, where each pair $(i, f\,i)$ is represented with i and with the payload of the stream at index i. The stream $[f\,0, f\,1, f\,2, \cdots[$ is said to *enumerate* the function f.

For example, in Scheme, one can implement a procedure `lazy-iota` that, when applied to an integer n, constructs the stream of payloads $[n, n+1, n+2, \cdots[$. One can also implement a procedure `lazy-map` that, given a procedure and a stream, applies this procedure to each payload in the stream and constructs the stream of the results, in the same order. Given a procedure `f` that implements the function $f : \mathbb{N} \to \mathbb{N}$, evaluating

```
(lazy-map f (lazy-iota 0))
```

yields a stream that enumerates f.

1.2.2 Differences and antidifferences

The prefix sum (a.k.a. antidifference) of a function $f : \mathbb{N} \to \mathbb{N}$ is $\nabla^{-1} f$:

$$\nabla^{-1} f = \lambda n. \textstyle\sum_{i=0}^{n} f\,i$$

It is used in Moessner's process (but we shall not use the ∇^{-1} notation).

The backward difference of a function $f : \mathbb{N} \to \mathbb{N}$ is ∇f:

$$\nabla f\,0 = f\,0 \quad \wedge \quad \forall n : \mathbb{N}, \nabla f\,(n+1) = f\,(n+1) - f\,n$$

It is used in the inverse of Moessner's process (Section 4.3).

∇^{-1} is a right inverse (as well as a left inverse) of ∇:

$$\begin{cases} \nabla\,(\nabla^{-1} f)\,0 & = \nabla\,(\lambda n. \sum_{i=0}^{n} f\,i)\,0 \\ & = \sum_{i=0}^{0} f\,i \\ & = f\,0 \\ \nabla\,(\nabla^{-1} f)\,(n+1) & = \nabla\,(\lambda n. \sum_{i=0}^{n} f\,i)\,(n+1) \\ & = (\lambda n. \sum_{i=0}^{n} f\,i)\,(n+1) - (\lambda n. \sum_{i=0}^{n} f\,i)\,n \\ & = f\,(n+1) \end{cases}$$

Also, for all expressions $e : \mathbb{N}$, the equality $\sum_{i=0}^{x} e = (x+1) \cdot e$ holds when the local variable i does not occur free in the expression e.

1.2.3 Dynamic programming

Dynamic programming [12] is a way of writing programs so that the results of overlapping local computations are memoized and then shared. This sharing ensures that the overlapping local computations are not repeated in the course of a global computation.

- Consider Fibonacci numbers, for example. They are inductively specified with the following second-order linear recurrence:

  $$F_0 = 0 \quad \wedge \quad F_1 = 1 \quad \wedge \quad \forall n : \mathbb{N}, F_{n+2} = F_n + F_{n+1}$$

  A recursive function mapping $n : \mathbb{N}$ to F_n can be implemented based on this inductive specification. As is well known, this additive implementation gives rise to overlapping intermediate computations in the form of many repeated identical recursive calls.

  In contrast, using dynamic programming, one can implement an additive program that iterates over successive pairs of Fibonacci numbers, starting from (F_0, F_1) and without repeating any computation. Given a number $n : \mathbb{N}$, the result F_n is found in one of the components of the resulting pair.

  For example, it is known since Burstall and Darlington [6] how to map (F_0, F_1) to (F_n, F_{n+1}) in n tail-recursive calls to a local procedure `visit`:

```
(define fib
  (lambda (n)
    (if (< n 0)
        (errorf 'fib "negative input: ~s" n)
        (letrec ([visit (lambda (i x y)
                          (if (= i 0)
                              x
                              (visit (- i 1) y (+ x y))))])
          (visit n 0 1)))))
```

  The Scheme procedure `fib` is applied to an integer n, verifies that this integer is non-negative, and iterates over successive triples of integers – a decreasing counter and two increasing Fibonacci numbers: it starts with n, 0, and 1 and continues with $n-1$, 1, and 1 and then with $n-2$, 1, and 2 and then with $n-3$, 2, and 3, until `visit` is applied to 0, F_n, and F_{n+1}. At that point, the iteration stops and F_n is returned.

  Dynamic programming is at work here because the structure of the overlapping computations makes it possible to compute each intermediate Fibonacci number only once. This structure can be exploited in many ways. For example, programs written using dynamic programming need not only be additive: a dynamic program that uses multiplications and divisions can compute Fibonacci numbers with a logarithmic rather than linear complexity.

- For another example, consider binomial coefficients. They are inductively specified as follows (the third clause is Pascal's rule): $\forall n : \mathbb{N}$,

  $$\binom{n}{0} = 1 \quad \wedge \quad \binom{n}{n} = 1 \quad \wedge \quad \forall k : \mathbb{N}, k < n, \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

  A recursive function mapping $n : \mathbb{N}$ and $k : \mathbb{N}$, where $k \leq n$, to $\binom{n}{k}$ can be implemented based on this inductive specification. As is well known, this additive implementation gives rise to overlapping intermediate computations in the form of many repeated identical recursive calls.

  In contrast, using dynamic programming, one can implement an additive program that iterates over successive lists of binomial coefficients, starting from $[\binom{0}{0}]$ and without repeating any computation. Given a number $n : \mathbb{N}$, the result $\binom{n}{k}$, for any $k : \mathbb{N}$ such that $k \leq n$, is found at index k in the resulting list $[\binom{n}{0}, \binom{n}{1}, \cdots, \binom{n}{n-1}, \binom{n}{n}]$. These successive lists form Pascal's triangle, an early example of dynamic programming.

The art of dynamic programming is to start from the recursive function that implements an inductive specification (e.g., of Fibonacci numbers or of binomial coefficients), to identify the overlapping computations, and to devise a process to compute intermediate results only once to obtain the desired result. The present article is about the converse: start from the computational process and reverse-engineer a recursive function. The starting point here is that Moessner's process of constructing successive streams fits the pattern of dynamic programming, and the point of Section 3 is to reverse-engineer a recursive function that implements an inductive specification of integral powers. The thesis defended here is that the essence of Moessner's theorem is in this inductive specification, not in the dynamic-programming infrastructure of Moessner's process.

2 Background and Related Work

Originally [35], Moessner's process started with $[1, 2, 3, \cdots[$. Paasche [46] pointed out that it could start with $[1, 1, 1, \cdots[$, which has the effect of undoing one iteration of the process. Later on [8], the author observed that one more iteration could be undone and that the process could start with $[1, 0, 0, \cdots[$. This observation is used in Section 1 for presentational purposes and revisited in Section 7.3 to recast Long's second theorem [29]. In the rest of this article, the process starts with $[1, 1, 1, \cdots[$, as per Paasche's suggestion, which shortcuts the initial iteration from $[1, 0, 0, \cdots[$ to $[1, 1, 1, \cdots[$.

Let us illustrate Moessner's process for the exponents 0, 1, 2, and 3.

Exponent 0: No iterations take place.

The result is the starting stream $[1, 1, 1, \cdots[$, i.e., $[1^0, 2^0, 3^0, \cdots[$, the stream of the positive integers exponentiated with 0.

Exponent 1: One iteration takes place.

(0) Filtering out each second element of $[1, 1, 1, \cdots[$ vacuously constructs $[1, 1, 1, \cdots[$. The stream of the resulting prefix sums is $[1, 2, 3, \cdots[$, i.e., $[1^1, 2^1, 3^1, \cdots[$, the stream of the positive integers exponentiated with 1.

Exponent 2: Two iterations take place.

(1) Filtering out each third element of $[1, 1, 1, \cdots[$ vacuously constructs $[1, 1, 1, \cdots[$. The stream of the resulting prefix sums is $[1, 2, 3, \cdots[$.

(0) Filtering out each second element of this last stream constructs $[1, 3, 5, \cdots[$, the stream of the positive odd integers. As mentioned at the bottom of page 58, the stream of the resulting prefix sums is $[1, 4, 9, \cdots[$, i.e., $[1^2, 2^2, 3^2, \cdots[$, the stream of the positive integers exponentiated with 2.

Exponent 3: Three iterations take place.

(2) Filtering out each fourth element of $[1, 1, 1, \cdots[$ vacuously constructs $[1, 1, 1, \cdots[$. The stream of the resulting prefix sums is $[1, 2, 3, \cdots[$.

(1) Filtering out each third element of this last stream constructs $[1, 2, 4, 5, 7, 8, 10, \cdots[$. The stream of the resulting prefix sums is $[1, 3, 7, 12, 19, 27, 37, \cdots[$.

(0) Filtering out each second element of this last stream constructs $[1, 7, 19, 37, 61, 91, \cdots[$. The stream of the resulting prefix sums is $[1, 8, 27, \cdots[$, i.e., $[1^3, 2^3, 3^3, \cdots[$, the stream of the positive integers exponentiated with 3.

Exponent 4: Four iterations take place.

(3) Filtering out each fifth element of $[1, 1, 1, \cdots[$ vacuously constructs $[1, 1, 1, \cdots[$. The stream of the resulting prefix sums is $[1, 2, 3, \cdots[$.

(2) Filtering out each fourth element of this last stream constructs $[1, 2, 3, 5, 6, 7, 9, \cdots[$. The stream of the resulting prefix sums is $[1, 3, 6, 11, 17, 24, 33, \cdots[$.

(1) Filtering out each third element of this last stream constructs $[1, 3, 11, 17, 33, 43, \cdots[$. The stream of the resulting prefix sums is $[1, 4, 15, 32, 65, 108, 175, 256, 369, 625, \cdots[$.

(0) Filtering out each second element of this last stream constructs $[1, 15, 65, 175, 369, \cdots[$. The stream of the resulting prefix sums is $[1, 16, 81, 256, \cdots[$, i.e., $[1^4, 2^4, 3^4, 4^4, \cdots[$, the stream of the positive integers exponentiated with 4.

In 1951 [35], Alfred Moessner presented this process and conjectured that given a positive integer n, it always ends with $[1^n, 2^n, 3^n, \cdots[$. This conjecture was then proved and is now referred to as Moessner's theorem.[1]

2.1 In the 20th Century

Oskar Perron (1) characterized the series of integers after the first periodic elision in Moessner's process and (2) characterized the series of integers after the next periodic elision, given the series of integers after a periodic elision. He expressed this series of integers in closed form and proved this closed form using mathematical induction. Then, using a telescoping sum (like at the top of page 59), he proved that the prefix sums of this series of integers compute powers, as conjectured by Moessner [50].

Ivan Paasche obtained what is now known as Moessner's theorem as a corollary of a more general theorem about generating functions [46] and then later using linear transformations [49]. He also pointed out the connection between Moessner's process and Pascal's triangle as well as Taylor series [47].

Jan van Yzeren documented a correspondence between numbers obtained in the course of Moessner's process and numbers obtained in the course of Horner's algorithm [61].

Hans Salié characterized the output of Moessner's process $[a_1^{(n)}, a_2^{(n)}, a_3^{(n)}, \cdots[$ in terms of the initial sequence $[a_1^{(1)}, a_2^{(1)}, a_3^{(1)}, \cdots[$ for any exponent $n \geq 2$ [54]. This characterization, which is not solely additive and is renamed here for notational consistency, reads as follows for any index $k \geq 1$ in the stream:

$$a_k^{(n)} = \sum_{i=0}^{k-1} \sum_{j=1}^{n-1} a_{i \cdot n+j}^{(1)} \cdot (k-i)^{n-1-j} \cdot (k-1-j)^{j-1}$$

Salié then pointed out that if $a_k^{(1)}$ is k then $a_k^{(n)}$ is k^n, as in Moessner's process, and that if $a_1^{(1)}$ is 1 and for all positive k, $a_{k+1}^{(1)}$ is 0, then $a_k^{(n)}$ is k^{n-2}, which anticipates the observation used in Section 1 for

[1] Alfred Moessner (1893–198?) was a German mathematics teacher who, besides regular research activities [37], contributed to recreational-mathematics journals like Scripta Mathematica (between 1936 and 1956 with 45 contributions) and Sphinx (between 1931 and 1933 with 16 contributions and then in 1939 with 1 contribution) with many results – diophantine equations, properties of integers, Pythagorean numbers, and a number of interesting identities. He is noted, e.g., for triplets [19, 33] as well as for magic squares [34], three of them solely containing prime numbers [36]: Honsberger refers to Moessner's results in arithmetic as "gems" [24, page 269]. Both in Scripta Arithmetica and in Sphinx, several of Moessner's "Curiosa" were followed by an insightful note from the Editor-in-Chief (Jekuthiel Ginsburg and Maurice Kraitchik, resp.), a testament of the liveliness of mathematics in the middle of the 20th century. (Scripta Arithmetica was reporting events as they were happening at the time, from the creation of the Journal of Symbolic Logic to the publication of Littlewood's miscellany, which makes it a fascinating – and sometimes heart-breaking – reading today.) As for his affiliation, Moessner did not mention any, simply listing Nürnberg in 1936 in his first contribution to Scripta Arithmetica, and then mentioning nothing until after the war, where he listed Gunzenhausen next to his name, as he did in the article that introduced his eponymous theorem [35].

presentational purposes and revisited in Section 7.3 to recast Long's second theorem [29]. Naturally, if for all positive k, $a_k^{(1)}$ is 1, then $a_k^{(n)}$ is k^{n-1}.

Using a generalization of Pascal's triangle and its binomial coefficients, Calvin Long initialized the process with a stream that follows an arithmetic progression [29], as revisited in Section 7.3. He extended his study by varying the period of elision [32], building on Paasche's observation that increasing this period makes the process map additions to multiplications, subtractions to quotients, and multiplications to exponentiations, as in a slide rule [10, 47, 48]. On this basis, he used Moessner's theorem to imaginatively motivate students in high school [30, 31, 56] – which is easier to achieve today thanks to the On-Line Encyclopedia of Integer Sequences [45]: the students can copy-paste the prefix of a stream and excitedly see whether it corresponds to a known sequence of integers.

Ross Honsberger presented a geometrical proof due to Karel Post [24].

2.2 In the 21st Century

After the turn of the century, Moessner's process became a classical exercise, and Roland Backhouse introduced Ralf Hinze to it as such [22]. Hinze elegantly presented a calculational proof of Moessner's theorem and then developed a comprehensive framework for streams [23]. Milad Niqui and Jan Rutten proved Moessner's theorem coalgebraically [38]. Robbert Krebbers, Louis Parlant, and Alexandra Silva formalized a coinduction proof of Moessner's theorem in Coq [27]. Dexter Kozen and Alexandra Silva vastly generalized Moessner's theorem using power series and presented an algebraic proof for it [26], which Mark Bickford formalized in the Nuprl proof assistant, the first formalization of Moessner's theorem and of its proof [3]. In his MSc thesis [58], Peter Urbak dualized Moessner's process, generating triangles column by column instead of row by row and singling out a collection of new properties. He also formalized and generalized the correspondence documented by van Yzeren between numbers obtained in the course of Moessner's process and numbers obtained in the course of Horner's algorithm [61]. In his BSc thesis [57], Uladzimir Treihis formalized Perron and Salié's proofs and generalized Salié's result to a version of Moessner's process without striking-out phase.

The author's motivation is the same as in his contribution to Glynn Winskel's Festschrift [8]. To quote:

> All in all, it seems to us that like Stonehenge, Moessner's theorem is like a mirror – every publication about it reflects what is in the mind of its authors: a property, a proof technique, a corollary, or a showcase for a framework.[2] Accordingly, the present article reflects what is in our mind as computer scientists and functional programmers: we want to program Moessner's [process] to understand not just how it works but also why it works.[3]

3 Moessner's Process Without Dynamic Programming

In Moessner's process, each of the streams enumerates a function. For example, $[1^n, 2^n, 3^n, \cdots[$ enumerates $\lambda x.(x+1)^n$. The goal of this section is to analyze Moessner's process (Section 3.1) and to recast its constitutive steps – namely the filtering-out phase (Section 3.2) and then the prefix sums (Section 3.3) – in terms of these enumerated functions instead of in terms of the enumerating streams. The result will be a streamless version of Moessner's process, i.e., a version without dynamic programming (Section 3.4).

[2] Or, like, dynamic programming?

[3] At the time, the author put forward the word "sieve" instead of "process" but this word is inaccurate since Moessner's process generates new numbers whereas a sieve does not.

3.1 Analysis

Let us characterize the stream processing that underlies Moessner's process. The filtering-out phase mentions each $i+1$st element of a stream: the index x of each second element has the property that $x \bmod 2 = 1$, the index x of each third element has the property that $x \bmod 3 = 2$, and for all $j : \mathbb{N}$, the index x of each $j+1$st element has the property that $x \bmod (j+1) = j$. The corresponding predicate reads $\lambda j.\lambda x.x \bmod (j+1) = j$, witness the following truth table,

$j \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	$\ldots$
1	F	T	F	T	F	T	F	T	F	T	F	T	F	T	
2	F	F	T	F	F	T	F	F	T	F	F	T	F	F	
3	F	F	F	T	F	F	F	T	F	F	F	T	F	F	
$\vdots$															

where T denotes true and F false. And indeed, e.g., in the row where $j = 2$, the index of each third element of the stream satisfies the predicate and the other indices do not. The salient point of this table is that its rows, read bottom up, characterize both the numbers filtered by Moessner's process and their period of elision.

3.2 The Filtering-Out Phase

For any given positive integer i, how do we filter out each $i+1$st element of a stream to construct a new stream? Put more simply, for $f : \mathbb{N} \to \mathbb{N}$, and given the stream $[f\,0, f\,1, f\,2, \cdots[$,

- how can $[f\,0, f\,2, f\,4, f\,6, f\,8, f\,10, f\,12, \cdots[$ be constructed, where each second element has been filtered out?
- how can $[f\,0, f\,1, f\,3, f\,4, f\,6, f\,7, f\,9, \cdots[$ be constructed, where each third element has been filtered out?
- etc.

To this end, let us define a function $g : \mathbb{N}^+ \to \mathbb{N} \to \mathbb{N}$ that is given the (positive) period of elision and the original input for f:

- $[f\,(g\,1\,0), f\,(g\,1\,1), f\,(g\,1\,2), \cdots[= [f\,0, f\,2, f\,4, f\,6, f\,8, f\,10, f\,12, \cdots[$
- $[f\,(g\,2\,0), f\,(g\,2\,1), f\,(g\,2\,2), \cdots[= [f\,0, f\,1, f\,3, f\,4, f\,6, f\,7, f\,9, f\,10, \cdots[$
- etc.

We define g so that the arithmetic progression is maintained and a number is periodically skipped. The definition of g hinges on integer division (and so we omit the floor notation used in the executive summary since we are considering integers, not rational numbers):

Property 1 (basic arithmetic for natural numbers)**.** $\forall j : \mathbb{N}, \forall x : \mathbb{N}$,

$$\begin{cases} x \bmod (j+1) < j \Rightarrow \frac{x+1}{j+1} = \frac{x}{j+1} \\ x \bmod (j+1) = j \Rightarrow \frac{x+1}{j+1} = \frac{x}{j+1} + 1 \end{cases}$$

Definition 1 (Elision function)**.** $g : \mathbb{N}^+ \to \mathbb{N} \to \mathbb{N} = \lambda j.\lambda x.\frac{(j+1)\cdot x}{j}$

Equivalently, we could define $g : \mathbb{N} \to \mathbb{N} \to \mathbb{N} = \lambda j.\lambda x.\frac{(j+2)\cdot x}{j+1} = \lambda j.\lambda x.x + \frac{x}{j+1}$, which illustrates the relevance of Property 1.

The following table illustrates g (in each row, $\circ$ prefixes the successor of a number that has been skipped):

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	$\ldots$
$g\,1\,x$	0	$\circ 2$	$\circ 4$	$\circ 6$	$\circ 8$	$\circ 10$	$\circ 12$	$\circ 14$	$\circ 16$	$\circ 18$	$\circ 20$	$\circ 22$	$\circ 24$	$\circ 26$	
$g\,2\,x$	0	1	$\circ 3$	4	$\circ 6$	7	$\circ 9$	10	$\circ 12$	13	$\circ 15$	16	$\circ 18$	19	
$g\,3\,x$	0	1	2	$\circ 4$	5	6	$\circ 8$	9	10	$\circ 12$	13	14	$\circ 16$	17	
$\vdots$															

And indeed, in the row for $g\,1\,x$, each successive second element has been filtered out, in the row for $g\,2\,x$, each successive third element has been filtered out, etc.

It also makes sense to define the complement of g – noted $\overline{g}$ – explicitly, since Moessner's process is defined in terms of the numbers that are filtered out, not in terms of the numbers that are filtered in [31,56]:

Definition 2 (Complement of the elision function)**.** $\overline{g} = \lambda j.\lambda x.(j+1)\cdot x + j$

Given a period of elision, the numbers that are skipped enumerate $\overline{g}$:

x	0	1	2	3	4	5	$\ldots$
$\overline{g}\,1\,x$	1	3	5	7	9	11	
$\overline{g}\,2\,x$	2	5	8	11	14	17	
$\overline{g}\,3\,x$	3	7	11	15	19	23	
$\vdots$							

3.3 The Prefix Sums

If a stream enumerates a given function f, then the stream of its prefix sums enumerates $\lambda x.\sum_{i=0}^{x} f\,i$.

3.4 Moessner's Process, Streamlessly

Instead of considering the successive streams, we consider the successive functions that these streams enumerate. So, given an exponent n, we start from the constant function $f_n = \lambda x.1$ rather than from the stream of 1's. The first iteration constructs $f_{n-1} = \lambda x.\sum_{i_n=0}^{x} f_n\,(g\,n\,i_n)$, the second iteration constructs $f_{n-2} = \lambda x.\sum_{i_{n-1}=0}^{x} f_{n-1}\,(g\,(n-1)\,i_{n-1})$, etc., and we shall unfold the definition of g as we go.

To illustrate, here are the renditions of Moessner's process for the exponents 0, 1, 2, 3, and 4, an echo of Section 2. The initial stream is $[1, 1, 1, \cdots[$ and we shall unfold the definitions of f_1, f_2, f_3, and f_4 as we go:

Exponent 0: The initial stream enumerates $f_0 = \lambda x.1$ and no iteration takes place. The result is $\lambda x.1$, i.e., $\lambda x.(x+1)^0$.

Exponent 1: The initial stream enumerates $f_1 = \lambda x.1$ and one iteration takes place.

(0) The iteration constructs $[1, 2, 3, \cdots[$. This stream enumerates $f_0 = \lambda x.\sum_{i_1=0}^{x} f_1\,(g\,1\,i_1) = \lambda x.\sum_{i_1=0}^{x} 1 = \lambda x.(x+1)$, i.e., $\lambda x.(x+1)^1$.

Exponent 2: The initial stream enumerates $f_2 = \lambda x.1$, and two iterations take place.

(1) The first iteration constructs $[1, 2, 3, \cdots[$. This stream enumerates $f_1 = \lambda x.\sum_{i_2=0}^{x} f_2\,(g\,2\,i_2) = \lambda x.\sum_{i_2=0}^{x} 1 = \lambda x.(x+1)$.

(0) The second iteration constructs $[1, 4, 9, \cdots[$. This stream enumerates $f_0 = \lambda x.\sum_{i_1=0}^{x} f_1\,(g\,1\,i_1) = \lambda x.\sum_{i_1=0}^{x} f_1\,(\frac{2\cdot i_1}{1}) = \lambda x.\sum_{i_1=0}^{x} (\frac{2\cdot i_1}{1}+1) = \lambda x.\sum_{i_1=0}^{x} (2\cdot i_1+1)$, i.e., $\lambda x.(x+1)^2$ since as detailed at the bottom of page 58 and at the top of page 59, summing the first odd natural numbers gives a square number.

Exponent 3: The initial stream enumerates $f_3 = \lambda x.1$, and three iterations take place.

(2) The first iteration constructs $[1, 2, 3, \cdots[$. This stream enumerates $f_2 = \lambda x.\sum_{i_3=0}^{x} f_3\,(g\,3\,i_3) = \lambda x.\sum_{i_3=0}^{x} 1 = \lambda x.(x+1)$.

(1) The second iteration constructs $[1, 3, 7, 12, 19, 27, 37, \cdots[$. This stream enumerates $f_1 = \lambda x.\sum_{i_2=0}^{x} f_2\,(g\,2\,i_2) = \lambda x.\sum_{i_2=0}^{x} f_2\,(\frac{3\cdot i_2}{2}) = \lambda x.\sum_{i_2=0}^{x} (\frac{3\cdot i_2}{2}+1)$.

(0) The third iteration constructs $[1, 8, 27, 64, \cdots[$. This stream enumerates $f_0 = \lambda x.\sum_{i_1=0}^{x} f_1\,(g\,1\,i_1) = \lambda x.\sum_{i_1=0}^{x} f_1\,(\frac{2\cdot i_1}{1}) = \lambda x.\sum_{i_1=0}^{x}\sum_{i_2=0}^{2\cdot i_1} (\frac{3\cdot i_2}{2}+1)$, and
$\sum_{i_1=0}^{x}\sum_{i_2=0}^{2\cdot i_1} (\frac{3\cdot i_2}{2}+1) = \sum_{i_1=0}^{x}\sum_{i_2=0}^{2\cdot i_1} (i_2+\frac{i_2}{2}+1) =$
$\sum_{i_1=0}^{x} (\sum_{i_2=0}^{2\cdot i_1} i_2 + \sum_{i_2=0}^{2\cdot i_1} \frac{i_2}{2} + \sum_{i_2=0}^{2\cdot i_1} 1) = \sum_{i_1=0}^{x} (\frac{(2\cdot i_1)\cdot(2\cdot i_1+1)}{2} + i_1^2 + (2\cdot i_1+1)) =$ (see Appendix C)
$\sum_{i_1=0}^{x} (3\cdot i_1^2 + 3\cdot i_1 + 1) = \sum_{i_1=0}^{x} ((i_1+1)^3 - i_1^3) = (x+1)^3$

Exponent 4: The initial stream enumerates $f_4 = \lambda x.1$, and four iterations take place.

(3) The first iteration constructs $[1, 2, 3, \cdots[$. This stream enumerates $f_3 = \lambda x.\sum_{i_4=0}^{x} f_4\,(g\,4\,i_4) = \lambda x.\sum_{i_4=0}^{x} 1$.

(2) The second iteration constructs $[1, 3, 6, 11, 17, 24, 33, \cdots[$. This stream enumerates $f_2 = \lambda x.\sum_{i_3=0}^{x} f_3\,(g\,3\,i_3) = \lambda x.\sum_{i_3=0}^{x} f_3\,(\frac{4\cdot i_3}{3})$.

(1) The third iteration constructs $[1, 4, 15, 32, 65, 108, 175, 256, \cdots[$. This stream enumerates $f_1 = \lambda x.\sum_{i_2=0}^{x} f_2\,(g\,2\,i_2) = \lambda x.\sum_{i_2=0}^{x} f_2\,(\frac{3\cdot i_2}{2})$.

(0) The fourth iteration constructs $[1, 16, 81, 256, \cdots[$. This stream enumerates $f_0 = \lambda x.\sum_{i_1=0}^{x} f_1\,(g\,1\,i_1) = \lambda x.\sum_{i_1=0}^{x} f_1\,(\frac{2\cdot i_1}{1}) = \ldots = \lambda x.\sum_{i_1=0}^{x} ((i_1+1)^4 - i_1^4)$, i.e., $\lambda x.(x+1)^4$.

Let us unfold all the functions that are enumerated by the intermediate streams for the exponents 0, 1, 2, 3, and 4 as well as 5:

$$\lambda x.1$$
$$\lambda x.\sum_{i_1=0}^{x} 1$$
$$\lambda x.\sum_{i_1=0}^{x}\sum_{i_2=0}^{g\,1\,i_1} 1$$
$$\lambda x.\sum_{i_1=0}^{x}\sum_{i_2=0}^{g\,1\,i_1}\sum_{i_3=0}^{g\,2\,i_2} 1$$
$$\lambda x.\sum_{i_1=0}^{x}\sum_{i_2=0}^{g\,1\,i_1}\sum_{i_3=0}^{g\,2\,i_2}\sum_{i_4=0}^{g\,3\,i_3} 1$$
$$\lambda x.\sum_{i_1=0}^{x}\sum_{i_2=0}^{g\,1\,i_1}\sum_{i_3=0}^{g\,2\,i_2}\sum_{i_4=0}^{g\,3\,i_3}\sum_{i_5=0}^{g\,4\,i_4} 1$$

Each nested sum is an instance of a prefix sum and each upper bound that involves g is an instance of the filtering-out phase. (In Section 7.4 and onwards, we consider other definitions of g, i.e., other filtering-out strategies.)

More generally, unfolding all the intermediate definitions leads one to the following streamless instances of Moessner's theorem for the exponents 0, 1, 2, 3, and 4, without the dynamic-programming infrastructure:

$$
\begin{array}{rcll}
\lambda x.(x+1)^0 & = & \lambda x.1 & \\
\lambda x.(x+1)^1 & = & \lambda x.\sum_{i_1=0}^{x} 1 & \\
\lambda x.(x+1)^2 & = & \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} 1 & = \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+\frac{i_1}{1}} 1 \\
\lambda x.(x+1)^3 & = & \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} 1 & = \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+\frac{i_1}{1}} \sum_{i_3=0}^{i_2+\frac{i_2}{2}} 1 \\
\lambda x.(x+1)^4 & = & \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} \sum_{i_4=0}^{\frac{4 \cdot i_3}{3}} 1 & = \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+\frac{i_1}{1}} \sum_{i_3=0}^{i_2+\frac{i_2}{2}} \sum_{i_4=0}^{i_3+\frac{i_3}{3}} 1
\end{array}
$$

4 A Left Inverse of Moessner's Process Without Dynamic Programming

Connecting prefix sums (antidifferences) and backward differences makes it possible to devise a left inverse for Moessner's process. This left inverse has no computational interest in terms of its final result – which is $[1^0, 2^0, 3^0, \cdots[$, i.e., $[1, 1, 1, \cdots[$, for any given exponent n and stream $[1^n, 2^n, 3^n, \cdots[$ – but studying it provides a second way to construct the intermediate streams. The fun is in the game but the reader in a hurry can safely skip this consolidating section.

4.1 Inverting the Prefix Sums

We use backward differences, since as shown in Section 1.2, they invert prefix sums. When going from $[1^0, 2^0, 3^0, \cdots[$ to $[1^n, 2^n, 3^n, \cdots[$ in Moessner's process, the exponents increase since each iteration of the process uses an antidifference, which is the discrete counterpart of an integration. And when going from $[1^n, 2^n, 3^n, \cdots[$ to $[1^0, 2^0, 3^0, \cdots[$ in the converse of Moessner's process, the exponents decrease since each iteration of the process uses a difference, which is the discrete counterpart of a differentiation.

4.2 Inverting the Filtering-Out Phase

To invert the filtering-out phase, we use a converse of the elision function from Section 3.2. The following table illustrates the idea of this converse with

$$h = \lambda j.\lambda x.\text{if } x \bmod (j+1) = j \text{ then } \square \text{ else } j \cdot \frac{x}{j+1}$$

where $\square$ is a placeholder for the stream element to insert and using the converse of $\frac{(j+1) \cdot x}{j}$, which yields the following "staircase" table:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
$h\,1\,x$	0	$\Box$	1	$\Box$	2	$\Box$	3	$\Box$	4	$\Box$	5	$\Box$	6	$\Box$	7	$\Box$	
$h\,2\,x$	0	0	$\Box$	2	2	$\Box$	4	4	$\Box$	6	6	$\Box$	8	8	$\Box$	10	
$h\,3\,x$	0	0	0	$\Box$	3	3	3	$\Box$	6	6	6	$\Box$	9	9	9	$\Box$	
$h\,4\,x$	0	0	0	0	$\Box$	4	4	4	4	$\Box$	8	8	8	8	$\Box$	12	
$\vdots$																	

And indeed:

- in the row for $h\,1\,x$, each stair has size 1 and every second cell, there is a placeholder between each stair;
- in the row for $h\,2\,x$, each stair has size 2 and every third cell, there is a placeholder between each stair,
- in the row for $h\,3\,x$, each stair has size 3 and every fourth cell, there is a placeholder between each stair,
- etc.

The converse of the filtering-out phase is achieved by replacing the stairs in the table above (e.g., 0, 0, 0 and 3, 3, 3 in the row for $h\,3\,x$) by increasing sequences (namely 0, 1, 2 and 3, 4, 5 in this row). These stairs are generated by the alternative branch, $j \cdot \frac{x}{j+1}$, i.e., when $x \bmod (j+1) \neq j$ in the definition of h. So we add $x \bmod (j+1)$ in that branch, which yields $k =$
$\lambda j.\lambda x.\text{if } x \bmod (j+1) = j \text{ then } \Box \text{ else } j \cdot \frac{x}{j+1} + x \bmod (j+1)$:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
$k\,1\,x$	0	$\Box$	1	$\Box$	2	$\Box$	3	$\Box$	4	$\Box$	5	$\Box$	6	$\Box$	7	$\Box$	
$k\,2\,x$	0	1	$\Box$	2	3	$\Box$	4	5	$\Box$	6	7	$\Box$	8	9	$\Box$	10	
$k\,3\,x$	0	1	2	$\Box$	3	4	5	$\Box$	6	7	8	$\Box$	9	10	11	$\Box$	
$k\,4\,x$	0	1	2	3	$\Box$	4	5	6	7	$\Box$	8	9	10	11	$\Box$	12	
$\vdots$																	

And indeed, each row contains the successive natural numbers if one skips the placeholders, which provides the indices for each of the intermediate streams in the converse of Moessner's process.

Earlier on [8], the author observed that in Moessner's process, the parts of the stream that are filtered out enumerate the successive monomials of the binomial expansion of $((x+1)+1)^n$ – which explains why the elements standing in the resulting stream enumerate $(x+1)^n$, an alternative proof of Moessner's theorem revisited in Section 5. So we splice in these successive monomials in the place of $\Box$ in the table above to invert the filtering-out phase.

4.3 Inverting Moessner's Process, Streamlessly

Here are the renditions of the left inverse of Moessner's process for the exponents 2, 3, and 4, as an echo of Section 3.4 and where $\omega_k = \lambda x.(x+1)^k$:

Exponent 2: The initial stream, $[1, 4, 9, \cdots[$, enumerates $f_0 = \omega_2$, i.e., $\lambda x.(x+1)^2$, or again $\lambda x.\binom{2}{2} \cdot (\frac{x}{1}+1)^2$, and two iterations take place.

(0) The first iteration constructs the stream $[1, 2, 3, \cdots[$. This stream enumerates $f_1 = \lambda x.\text{if } x \bmod 2 = 1 \text{ then } \binom{2}{1} \cdot (\frac{x}{2}+1)^1 \text{ else } \nabla f_0\,(1 \cdot \frac{x}{2} + x \bmod 2)$.

(1) The second iteration constructs the stream $[1, 1, 1, \cdots[$. This stream enumerates $f_2 = \lambda x.\text{if } x \bmod 3 = 2 \text{ then } \binom{2}{0} \cdot (\frac{x}{3}+1)^0 \text{ else } \nabla f_1\,(2 \cdot \frac{x}{3} + x \bmod 3)$.

Each f_i is equivalent to the corresponding f_i for exponent 2 in Section 3.4 and f_2 is equivalent to ω_0, i.e., $\lambda x.(x+1)^0$.

Exponent 3: The initial stream, $[1, 8, 27, 64, \cdots [$, enumerates $f_0 = \omega_3$, i.e., $\lambda x.(x+1)^3$, or again $\lambda x.\binom{3}{3} \cdot (\frac{x}{1}+1)^3$, and three iterations take place.

(0) The first iteration constructs the stream $[1, 3, 7, 12, 19, \cdots [$. This stream enumerates $f_1 = \lambda x.\text{if } x \bmod 2 = 1 \text{ then } \binom{3}{2} \cdot (\frac{x}{2}+1)^2 \text{ else } \nabla f_0\,(1 \cdot \frac{x}{2} + x \bmod 2)$.

(1) The second iteration constructs the stream $[1, 2, 3, \cdots [$. This stream enumerates $f_2 = \lambda x.\text{if } x \bmod 3 = 2 \text{ then } \binom{3}{1} \cdot (\frac{x}{3}+1)^1 \text{ else } \nabla f_1\,(2 \cdot \frac{x}{3} + x \bmod 3)$.

(2) The third iteration constructs the stream $[1, 1, 1, \cdots [$. This stream enumerates $f_3 = \lambda x.\text{if } x \bmod 4 = 3 \text{ then } \binom{3}{0} \cdot (\frac{x}{4}+1)^0 \text{ else } \nabla f_2\,(3 \cdot \frac{x}{4} + x \bmod 4)$.

Each f_i is equivalent to the corresponding f_i for exponent 3 in Section 3.4 and f_3 is equivalent to ω_0, i.e., $\lambda x.(x+1)^0$.

Exponent 4: The initial stream, $[1, 16, 81, 256, \cdots [$, enumerates $f_0 = \omega_4$, i.e., $\lambda x.(x+1)^4$, or again $\lambda x.\binom{4}{4} \cdot (\frac{x}{1}+1)^4$, and four iterations take place.

(0) The first iteration constructs the stream $[1, 4, 15, 32, \cdots [$. This stream enumerates $f_1 = \lambda x.\text{if } x \bmod 2 = 1 \text{ then } \binom{4}{3} \cdot (\frac{x}{2}+1)^3 \text{ else } \nabla f_0\,(1 \cdot \frac{x}{2} + x \bmod 2)$.

(1) The second iteration constructs the stream $[1, 3, 6, 11, \cdots [$ This stream enumerates $f_2 = \lambda x.\text{if } x \bmod 3 = 2 \text{ then } \binom{4}{2} \cdot (\frac{x}{3}+1)^2 \text{ else } \nabla f_1\,(2 \cdot \frac{x}{3} + x \bmod 3)$.

(2) The third iteration constructs the stream $[1, 2, 3, \cdots [$ This stream enumerates $f_3 = \lambda x.\text{if } x \bmod 4 = 3 \text{ then } \binom{4}{1} \cdot (\frac{x}{4}+1)^1 \text{ else } \nabla f_2\,(3 \cdot \frac{x}{4} + x \bmod 4)$.

(3) The fourth iteration constructs the stream $[1, 1, 1, \cdots [$. This stream enumerates $f_4 = \lambda x.\text{if } x \bmod 5 = 4 \text{ then } \binom{4}{0} \cdot (\frac{x}{5}+1)^0 \text{ else } \nabla f_3\,(4 \cdot \frac{x}{5} + x \bmod 5)$.

Each f_i is equivalent to the corresponding f_i for exponent 4 in Section 3.4 and f_4 is equivalent to ω_0, i.e., $\lambda x.(x+1)^0$.

All in all, we now have two ways to carry out the iterations of Moessner's process, namely forward and backward. These two ways consolidate our understanding of it.

5 The Essence of Moessner's Theorem

The key observation in Section 4.2 was that Moessner's process hinges on a family of equalities indexed by the exponent. Here are members of this family for the exponents 0, 1, 2, 3, and 4:

Exponent 0: $\forall i_0 : \mathbb{N}, 1 = \binom{0}{0} \cdot (i_0+1)^0$

Exponent 1:

$$\forall i_1 : \mathbb{N}, 1 = \binom{1}{0} \cdot (i_1+1)^0$$

$$\forall i_0 : \mathbb{N}, \textstyle\sum_{i_1=0}^{i_0} 1 = \binom{1}{1} \cdot (i_0+1)^1$$

Exponent 2:

$$\forall i_2 : \mathbb{N}, 1 = \binom{2}{0} \cdot (i_2+1)^0$$

$$\forall i_1 : \mathbb{N}, \textstyle\sum_{i_2=0}^{i_1} 1 = \binom{2}{1} \cdot (i_1+1)^1$$

$$\forall i_0 : \mathbb{N}, \textstyle\sum_{i_1=0}^{i_0} \sum_{i_2=0}^{g\,1\,i_1} 1 = \binom{2}{2} \cdot (i_0+1)^2$$

Exponent 3:
$$\forall i_3 : \mathbb{N}, 1 = \binom{3}{0} \cdot (i_3+1)^0$$
$$\forall i_2 : \mathbb{N}, \sum_{i_3=0}^{i_2} 1 = \binom{3}{1} \cdot (i_2+1)^1$$
$$\forall i_1 : \mathbb{N}, \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{g\,2\,i_2} 1 = \binom{3}{2} \cdot (i_1+1)^2$$
$$\forall i_0 : \mathbb{N}, \sum_{i_1=0}^{i_0} \sum_{i_2=0}^{g\,1\,i_1} \sum_{i_3=0}^{g\,2\,i_2} 1 = \binom{3}{3} \cdot (i_0+1)^3$$

Exponent 4:
$$\forall i_4 : \mathbb{N}, 1 = \binom{4}{0} \cdot (i_4+1)^0$$
$$\forall i_3 : \mathbb{N}, \sum_{i_4=0}^{i_3} 1 = \binom{4}{1} \cdot (i_3+1)^1$$
$$\forall i_2 : \mathbb{N}, \sum_{i_3=0}^{i_2} \sum_{i_4=0}^{g\,3\,i_3} 1 = \binom{4}{2} \cdot (i_2+1)^2$$
$$\forall i_1 : \mathbb{N}, \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{g\,2\,i_2} \sum_{i_4=0}^{g\,3\,i_3} 1 = \binom{4}{3} \cdot (i_1+1)^3$$
$$\forall i_0 : \mathbb{N}, \sum_{i_1=0}^{i_0} \sum_{i_2=0}^{g\,1\,i_1} \sum_{i_3=0}^{g\,2\,i_2} \sum_{i_4=0}^{g\,3\,i_3} 1 = \binom{4}{4} \cdot (i_0+1)^4$$

where the elision function g was defined at the bottom of page 65 in Section 3.2: $g = \lambda j.\lambda x.\frac{(j+1)\cdot x}{j} = \lambda j.\lambda x.x + \frac{x}{j}$. Ostensibly, the type of g is $\mathbb{N}^+ \to \mathbb{N} \to \mathbb{N}$, but in actuality, and as illustrated just above, its domain is $[1 .. n-1]$, where n is the given exponent. In words: starting from 1, a series of summations is performed that periodically filters in numbers with a period that starts at the desired exponent n and that shrinks at each iteration until it reaches 0. And at each iteration, the numbers that are filtered out sum up to a monomial in the binomial expansion of $((x+1)+1)^n$, where x is the upper bound in each outer summation.

Moessner's theorem is the last equality in each member of this family of equalities:

Exponent 0: $\forall x : \mathbb{N}, 1 \;=\; (x+1)^0$

Exponent 1: $\forall x : \mathbb{N}, \sum_{i_1=0}^{x} 1 \;=\; (x+1)^1$

Exponent 2: $\forall x : \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2\cdot i_1}{1}} 1 \;=\; (x+1)^2$

Exponent 3: $\forall x : \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2\cdot i_1}{1}} \sum_{i_3=0}^{\frac{3\cdot i_2}{2}} 1 \;=\; (x+1)^3$

Exponent 4: $\forall x : \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2\cdot i_1}{1}} \sum_{i_3=0}^{\frac{3\cdot i_2}{2}} \sum_{i_4=0}^{\frac{4\cdot i_3}{3}} 1 \;=\; (x+1)^4$

Exponent 5: $\forall x : \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+\frac{i_1}{1}} \sum_{i_3=0}^{i_2+\frac{i_2}{2}} \sum_{i_4=0}^{i_3+\frac{i_3}{3}} \sum_{i_5=0}^{i_4+\frac{i_4}{4}} 1 \;=\; (x+1)^5$

$\vdots$

And so independently of dynamic programming, the essence of Moessner's theorem is nested summations (as many nested summations as the degree of the result) with fractionally increasing upper bounds:

Theorem 1 (Moessner's theorem without dynamic programming)**.** $\forall x : \mathbb{N}, \forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2\cdot i_1}{1}} \sum_{i_3=0}^{\frac{3\cdot i_2}{2}} \cdots \sum_{i_n=0}^{\frac{n\cdot i_{n-1}}{n-1}} 1 \;=\; \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+\frac{i_1}{1}} \sum_{i_3=0}^{i_2+\frac{i_2}{2}} \cdots \sum_{i_n=0}^{i_{n-1}+\frac{i_{n-1}}{n-1}} 1 \;=\; (x+1)^n$$

These nested summations involve many overlapping subcomputations. For example, $\sum_{i_1=0}^{x} \sum_{i_2=0}^{2\cdot i_1} 1 = \sum_{i_1=0}^{x-2} \sum_{i_2=0}^{2\cdot i_1} 1 + \sum_{i_2=0}^{2\cdot(x-1)} 1 + \sum_{i_2=0}^{2\cdot x} 1$ when $x \geq 2$. Since $\sum_{i_2=0}^{2\cdot x} 1 = \sum_{i_2=0}^{2\cdot x-2} 1+1+1 = \sum_{i_2=0}^{2\cdot(x-1)} 1+1+1$, the expression $\sum_{i_2=0}^{2\cdot(x-1)} 1$ is independently computed several times, an opportunity for dynamic programming that Section 8 revisits and quantifies.

In Moessner's theorem, the upper bound of each inner summation is a fractional increment of the current index. It is instructive to tune these upper bounds, i.e., to try other filtering-out strategies [14]:

- How about not incrementing the current index?

  Corollary 1 (Binomial coefficients). $\forall x : \mathbb{N}, \forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{i_2} \cdots \sum_{i_n=0}^{i_{n-1}} 1 \quad = \quad \binom{x+n}{n}$$

- How about incrementing the current index with 1?

  Corollary 2 (Catalan numbers). $\forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{0} \sum_{i_2=0}^{i_1+1} \sum_{i_3=0}^{i_2+1} \cdots \sum_{i_n=0}^{i_{n-1}+1} 1 \quad = \quad \frac{\binom{2\cdot n}{n}}{n+1} \;=\; C_n$$

  Generalizing the outer upper bound from 0 to a non-negative integer x gives rise – according to the On-Line Encyclopedia of Integer Sequences [45] – to the $x+1$st convolution of Catalan numbers:

  Corollary 3 (Catalan numbers, convolved). $\forall x : \mathbb{N}, \forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1+1} \sum_{i_3=0}^{i_2+1} \cdots \sum_{i_n=0}^{i_{n-1}+1} 1 \quad = \quad \frac{(x+1)\cdot\binom{2\cdot n+x}{n}}{n+x+1}$$

- How about adding the current index to all the previous indices?

  Corollary 4 (A125860). $\forall x : \mathbb{N}, \forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{x} \sum_{i_2=0}^{x+i_1} \sum_{i_3=0}^{x+i_1+i_2} \cdots \sum_{i_n=0}^{x+i_1+i_2+\ldots+i_{n-1}} 1$$

  yields the integer at Column x and row n in Table A125860 in the On-Line Encyclopedia of Integer Sequences.

- How about multiplying the current index with all the previous indices?

  Corollary 5 (Positive integers). $\forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{1} \sum_{i_2=0}^{1\cdot i_1} \sum_{i_3=0}^{1\cdot i_1\cdot i_2} \cdots \sum_{i_n=0}^{1\cdot i_1\cdot i_2\ldots\cdot i_{n-1}} 1 \quad = \quad n+1$$

- How about emulating Fibonacci numbers?

  Corollary 6 (A137273). $\forall n : \mathbb{N}$,

$$\sum_{i_1=0}^{0} \sum_{i_2=0}^{1} \sum_{i_3=0}^{i_1+i_2} \sum_{i_4=0}^{i_2+i_3} \cdots \sum_{i_n=0}^{i_{n-2}+i_{n-1}} 1$$

  yields the number of partitions of a Fibonacci number into Fibonacci parts according to the On-Line Encyclopedia of Integer Sequences.

All these corollaries were discovered by playing with parameterized versions of Moessner's process without dynamic programming. Section 6 presents an unparameterized version and Section 7 presents several parameterized versions.

6 The Essence of Moessner's Process

We are now in position to state the essence of Moessner's process with a summation function and without dynamic programming. Here is a recursive implementation of this summation function:

```
(define Sigma_rec
  (lambda (n f)
    (if (< n 0)
        (errorf 'Sigma_rec "negative index: ~s" n)  ;;; either that or 0
        (letrec ([visit (lambda (i)
                          (+ (if (= i 0)
                                 0
                                 (visit (- i 1)))
                             (f i)))])
          (visit n)))))
```

For example, given a function f implemented by a procedure denoted by `f`, evaluating `(Sigma_rec 3 f)` gives rise to evaluating `(+ (+ (+ (+ 0 (f 0)) (f 1)) (f 2)) (f 3))`.

In practice, we implement summation iteratively with an equivalent tail-recursive procedure that uses an accumulator:

```
(define Sigma
  (lambda (n f)
    (if (< n 0)
        (errorf 'Sigma "negative index: ~s" n)  ;;; either that or 0
        (letrec ([visit (lambda (i a)
                          (if (= i 0)
                              a
                              (let ([ip (- i 1)])
                                (visit ip (+ (f ip) a)))))])
          (visit n (f n))))))
```

Given a function f implemented by a procedure denoted by `f`, evaluating `(Sigma 3 f)` gives rise to evaluating `(+ (f 0) (+ (f 1) (+ (f 2) (f 3))))`, where addition is re-associated and no addition to 0 occurs [13].

Here is an implementation of g:

```
(define g
  (lambda (j x)
    (if (<= j 0)
        (errorf 'g "non-positive first argument: ~s" j)
        (quotient (* (+ j 1) x) j))))
```

And here is a recursive implementation of Moessner's process without dynamic programming:

```
(define moessner
  (lambda (x n)
    (if (< n 0)
        (errorf 'moessner "negative exponent: ~s" n)
        (letrec ([visit (lambda (i x j)
                          (if (= i 0)
                              1
                              (Sigma x (lambda (x)
                                         (visit (- i 1) (g j x) (+ j 1))))))])
          (visit n x 1)))))
```

We then recast Moessner's theorem to characterize the output of Moessner's process without dynamic programming:

Theorem 2 (Moessner's theorem without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner x n)
```

yields $(x+1)^n$.

In other words, the function implemented by the Scheme procedure

```
(lambda (x) (moessner x n))
```

is enumerated by $[1^n, 2^n, 3^n, \cdots[$.

7 Parameterized Implementations of Moessner's Process

Towards recasting Long's theorems (Sections 7.2 and 7.3), let us parameterize Moessner's process with a function to apply to the current variable in the base case:

```
(define moessner-init
  (lambda (x n init)
    (if (< n 0)
        (errorf 'moessner-init "negative exponent: ~s" n)
        (letrec ([visit (lambda (i x j)
                          (if (= i 0)
                              (init x)
                              (Sigma x (lambda (x)
                                         (visit (- i 1) (g j x) (+ j 1))))))])
          (visit n x 1)))))
```

Compared to `moessner`, `moessner-init` takes an extra `init` argument and applies it to the current `x` in the base case.

7.1 Moessner's Theorem

Instantiating `init` with the constant function that yields `1` gives the same exponentiation as in Section 6:

Corollary 7 (Moessner's theorem without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-init x n (lambda (x) 1))
```

yields $(x+1)^n$.

In this corollary, `(lambda (x) 1)` implements $\lambda x.1$, which is enumerated by $[1, 1, 1, \cdots[$.

Alternatively, one can also go for another round of summation:

Corollary 8 (Moessner's theorem with another round of summation)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-init x n (lambda (x) (Sigma x (lambda (x) 1))))
```

yields $(x+1)^{n+1}$.

In this corollary, `(lambda (x) (Sigma x (lambda (x) 1)))` implements $\lambda x.\sum_{i=0}^{x} 1$, which is enumerated by $[x+1, x+1, x+1, \cdots[$, for any given x.

7.2 Long's First Theorem

Long's first theorem [29] is about Moessner's process when it starts with a constant stream $[a, a, a, \cdots[$ – which enumerates $\lambda x.a$:

Corollary 9 (Long's first theorem without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n`, for all $x : \mathbb{N}$ denoted by `x`, and for all $a : \mathbb{N}$ denoted by `a`, evaluating

```
(moessner-init x n (lambda (x) a))
```

yields $a \cdot (x+1)^n$.

The phrasing of Moessner's theorem with nested sums in Section 3.4 makes Long's first theorem limpid. For example, the equality

$$\lambda x.a \cdot (x+1)^3 = \lambda x.\sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} a$$

holds because of the following factorization: $\forall x : \mathbb{N}$,

$$\sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} a = \sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} a \cdot 1 = a \cdot \sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} 1$$

7.3 Long's Second Theorem

Long's second theorem [29] is about Moessner's process when it starts with a stream that follows an arithmetic progression: $[a, a+d, a+2 \cdot d, a+3 \cdot d, \cdots[$. For this theorem, it is more telling to undo one step of the process and add one initial iteration, so that instead of starting with $[1, 1, 1, \cdots[$, Moessner's process starts with $[1, 0, 0, \cdots[$, as in the opening of Section 1:

```
(define moessner-init-plus
  (lambda (x n init)
    (if (< n 0)
        (errorf 'moessner-init-plus "negative exponent: ~s" n)
        (letrec ([visit (lambda (i x j)
                          (if (= i 0)
                              (init x)
                              (Sigma x (lambda (x)
                                         (visit (- i 1) (g j x) (+ j 1))))))])
          (visit (+ n 1) x 1)))))
```

Compared to `moessner-init`, the initial call to `visit` is over `(+ n 1)` instead of `n`, which provides one more iteration.

Moessner's theorem accounts for the initial stream $[1, 0, 0, \cdots[$ – which enumerates $\lambda x.$if $x = 0$ then 1 else 0:

Corollary 10 (Moessner without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-init-plus x n (lambda (x) (if (= x 0) 1 0)))
```

yields $(x+1)^n$.

After the new initial iteration, the function is enumerated by $[1, 1, 1, \cdots[$, as in Corollary 7.

Long's first theorem accounts for the initial stream $[a, 0, 0, \cdots[$ – which enumerates $\lambda x.$if $x = 0$ then a else 0:

Corollary 11 (Long's first theorem without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n`, for all $x : \mathbb{N}$ denoted by `x`, and for all $a : \mathbb{N}$ denoted by `a`, evaluating

```
(moessner-init-plus x n (lambda (x) (if (= x 0) a 0)))
```

yields $a \cdot (x+1)^n$.

After the new initial iteration, the function is enumerated by $[a, a, a, \cdots[$, as in Corollary 9.

Long's second theorem accounts for the initial stream $[a, d, d, \cdots[$ – which enumerates $\lambda x.\text{if } x = 0 \text{ then } a \text{ else } d$:

Corollary 12 (Long's second theorem without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n`, for all $x : \mathbb{N}$ denoted by `x`, for all $a : \mathbb{N}$ denoted by `a`, and for all $d : \mathbb{N}$ denoted by `d`, evaluating

```
(moessner-init-plus x n (lambda (x) (if (= x 0) a d)))
```

yields $(a + d \cdot x) \cdot (x + 1)^n$.

After the new initial iteration, the function is enumerated by $[a, a + d, a + 2 \cdot d, \cdots[$: Long's arithmetic progression is obtained by this new initial iteration.

7.4 A Corollary About the Additive Generation of Integral Powers

For all that the streamless version of Moessner's process presented in Section 3.4 and the essence of Moessner's theorem presented in Section 5 involve fewer concepts – namely only iterated summations and no streams – they do not shine a new light on Moessner's insight. Iterative summations seem intuitive enough, but where in the spheres did Moessner get the idea of filtering out and widening the range?

To (unsuccessfully at first, but please do read on) address this question, let us also parameterize our implementation of Moessner's process with g:

```
(define moessner-fold
  (lambda (x n init g)
    (if (< n 0)
        (errorf 'moessner-fold "negative exponent: ~s" n)
        (letrec ([visit (lambda (i x j)
                          (if (= i 0)
                              (init x)
                              (Sigma x (lambda (x)
                                         (visit (- i 1) (g j x) (+ j 1))))))])
          (visit n x 1)))))
```

Corollary 13 (Moessner's theorem without dynamic programming)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-fold x n (lambda (x) 1) g)
```

yields $(x + 1)^n$.

We are now in position to play with the starting point of the process and with, e.g., its period of elision, as per Long's program [32]. Here is another suggestion, though: how about defining a version of Moessner's process with a constant filtering-out phase? Because the result still computes integral powers:

Corollary 14 (Moessner's theorem without dynamic programming, simpler)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-fold x n (lambda (x) 1) (lambda (j _) x))
```

yields $(x + 1)^n$.

Inlining the definition of `moessner-fold` here gives rise to the following stolid instances of Moessner's theorem for the exponents 0, 1, 2, 3, and 4 where powers are generated additively, which was the name of the game:

$$\lambda x.(x+1)^0 = \lambda x.1 \qquad \lambda x.(x+1)^3 = \lambda x.\textstyle\sum_{i=0}^{x} \sum_{i=0}^{x} \sum_{i=0}^{x} 1$$

$$\lambda x.(x+1)^1 = \lambda x.\textstyle\sum_{i=0}^{x} 1 \qquad \lambda x.(x+1)^4 = \lambda x.\sum_{i=0}^{x} \sum_{i=0}^{x} \sum_{i=0}^{x} \sum_{i=0}^{x} 1$$

$$\lambda x.(x+1)^2 = \lambda x.\textstyle\sum_{i=0}^{x} \sum_{i=0}^{x} 1 \qquad \lambda x.(x+1)^5 = \lambda x.\sum_{i=0}^{x} \sum_{i=0}^{x} \sum_{i=0}^{x} \sum_{i=0}^{x} \sum_{i=0}^{x} 1$$

This version is immensely simpler to understand since as foreshadowed in Section 1.2.2, for all expressions $e : \mathbb{N}$, the equality $\sum_{i=0}^{x} e = (x+1) \cdot e$ holds when the local variable i does not occur free in the expression e. It is also immensely simpler to prove (a routine induction): $\forall x : \mathbb{N}$,

$$1 \;=\; (x+1)^0 \qquad \wedge \qquad \forall n : \mathbb{N}, \underbrace{\sum_{i=0}^{x} \sum_{i=0}^{x} \cdots \sum_{i=0}^{x}}_{n+1} 1 \;=\; (x+1) \cdot \underbrace{\sum_{i=0}^{x} \cdots \sum_{i=0}^{x}}_{n} 1$$

A data point: in both versions, experiments consistently show that the summations induce the *same* number of additions – namely $(x+1)^n - 1$, which is logical without memoization – to compute the resulting power if we do not count the increment performed by g as an addition. For example, not only does the following equality hold

$$\forall x : \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{\frac{2 \cdot i_1}{1}} \sum_{i_3=0}^{\frac{3 \cdot i_2}{2}} \sum_{i_4=0}^{\frac{4 \cdot i_3}{3}} \sum_{i_5=0}^{\frac{5 \cdot i_4}{4}} 1 \;=\; \sum_{i_1=0}^{x} \sum_{i_2=0}^{x} \sum_{i_3=0}^{x} \sum_{i_4=0}^{x} \sum_{i_5=0}^{x} 1$$

but the summations in the ethereal left-hand side and in the stolid right-hand side induce the same total number of additions, namely $(x+1)^5 - 1$, consistently with Church's λ definability exercise of expressing multiplication as iterated addition and exponentiation as iterated multiplication [7]. So Moessner's filtering out with $(j+1) \cdot x + j$ (in the definition of $\overline{g}$) while widening the range of the summation with $\frac{(j+1) \cdot x}{j}$ (in the definition of g) balance out. As just illustrated though, these two features – filtering out and widening the range of the summation – are unnecessary to compute integral powers additively.

In passing, the number of additions in the simpler version can be lowered logarithmically if multiplication is defined in terms of addition by successive divisions by 2:

$$\begin{array}{rcl} \mathit{times}\, 0\, c & = & 0 \\ \mathit{times}\, (2 \cdot (x+1))\, c & = & 2 \cdot \mathit{times}\, (x+1)\, c \\ \mathit{times}\, (2 \cdot x + 1)\, c & = & 2 \cdot \mathit{times}\, x\, c + c \end{array}$$

where $2 \cdot \mathit{times}\,(x+1)\,c$ is not computed as $\mathit{times}\,(x+1)\,c + \mathit{times}\,(x+1)\,c$ but with the strict function application $(\lambda m.m+m)\,(\mathit{times}\,(x+1)\,c)$ and likewise for $2 \cdot \mathit{times}\,x\,c$. (This strict function application was implemented as a let insertion in Similix [4].)

All told,

$$\forall x : \mathbb{N}, \; (x+1)^n \;=\; \underbrace{\mathit{times}\,(x+1)\,(\mathit{times}\,(x+1)\,(\cdots(\mathit{times}\,(x+1)}_{n}\; 1)\cdots)).$$

This formulation suggests a further improvement using the computational content of Kleene's S_n^m-theorem [25], i.e., partial evaluation [9]: (1) specialize *times* with respect to its first argument, and (2) use this specialized version to carry out the exponentiation.

One can then re-introduce the dynamic-programming infrastructure to obtain a stream-based process for generating powers additively that is simpler than Moessner's:

0	1	2	3	$\cdots$	indices in the streams
[1,	1,	1,	1,	$\cdots$[	stream that enumerates $\lambda x.(x+1)^0$
[1,	2,	3,	4,	$\cdots$[	stream that enumerates $\lambda x.(x+1)^1$
[1,	4,	9,	16,	$\cdots$[	stream that enumerates $\lambda x.(x+1)^2$
[1,	8,	27,	64,	$\cdots$[	stream that enumerates $\lambda x.(x+1)^3$

Given a stream of positive natural numbers exponentiated with n, each number at index i in the stream of positive natural numbers exponentiated with $n+1$ is obtained by applying $\sum_{x=0}^{i}$ to the number at index i in the stream of positive natural numbers exponentiated with n. For example, following the first diagonal, $2 = \sum_{x=0}^{1} 1$, $9 = \sum_{x=0}^{2} 3$, $64 = \sum_{x=0}^{3} 16$, etc. and each summation can be carried out in logarithmic time rather than in linear time, which makes this process more efficient than Moessner's.

Also, Long's first theorem still applies: instead of starting from 1, one can start from another integer and see the resulting integral power multiplied by this other integer.

All in all, this alternative additive computation of powers sheds a quantitative light on Moessner's process without memoization in that it lets us characterize its number of additions, which is the same as the number of additions when exponentiation is defined by iterated multiplication and multiplication is defined as iterated addition. But it does not shed a qualitative light on Moessner's idea of filtering out and widening, since as it turns out, this idea is not needed to compute integral powers additively.

7.5 A Corollary About the Additive Generation of Factorial Numbers

The literature contains many variations over Moessner's process that keep the idea of successively filtering out and computing prefix sums, chiefly by varying the period of elision in the filtering-out phase, but not only. Most noticeably, the rules of the game are changed in that the result is no longer the last stream, but the stream consisting in the first element of each intermediate stream [47], just like in Eratosthenes's sieve. In the literature [3, 10, 22, 23, 26, 29, 32], this path is first illustrated with generating Factorial numbers additively, but we refrain to take it here. We do note, however, that it is possible to generate Factorial numbers additively without changing the rules of the game:

Corollary 15 (Factorial numbers)**.** For all $n : \mathbb{N}$ denoted by `n`, evaluating

```
(moessner-fold 0 n (lambda (x) 1) (lambda (j x) j))
```

yields $n!$.

In other words, the function implemented by the Scheme procedure

```
(lambda (n) (moessner-fold 0 n (lambda (x) 1) (lambda (j x) j)))
```

is enumerated by the stream of Factorial numbers [40].

Again, inlining the definition of `moessner-fold` here gives rise to the following stolid instances of Moessner's theorem to generate Factorial numbers additively:

$$
\begin{array}{rcl}
1 & = & 0! \\
\sum_{i=0}^{0} 1 & = & \sum_{i=0}^{0} 0! \;=\; 1 \cdot 0! \;=\; 1! \\
\sum_{i=0}^{1} \sum_{i=0}^{0} 1 & = & \sum_{i=0}^{1} 1! \;=\; 2 \cdot 1! \;=\; 2! \\
\sum_{i=0}^{2} \sum_{i=0}^{1} \sum_{i=0}^{0} 1 & = & \sum_{i=0}^{2} 2! \;=\; 3 \cdot 2! \;=\; 3! \\
\sum_{i=0}^{3} \sum_{i=0}^{2} \sum_{i=0}^{1} \sum_{i=0}^{0} 1 & = & \sum_{i=0}^{3} 3! \;=\; 4 \cdot 3! \;=\; 4! \\
\sum_{i=0}^{4} \sum_{i=0}^{3} \sum_{i=0}^{2} \sum_{i=0}^{1} \sum_{i=0}^{0} 1 & = & 5 \cdot (4 \cdot (3 \cdot (2 \cdot (1 \cdot 1)))) \;=\; 5!
\end{array}
$$

This version is simple to understand and simple to prove (another routine induction):

$$\sum_{i=0}^{0} 1 \;=\; 1! \qquad \wedge \qquad \forall n : \mathbb{N},\; \underbrace{\sum_{i=0}^{n+1} \sum_{i=0}^{n} \cdots \sum_{i=0}^{0}}_{n+2} 1 \;=\; (n+2) \cdot \underbrace{\sum_{i=0}^{n} \cdots \sum_{i=0}^{0}}_{n+1} 1$$

We note that the intermediate results are rising Factorial numbers. These intermediate results could just as well be falling Factorial numbers:

Corollary 16 (Factorial numbers)**.** For all $n : \mathbb{N}$ denoted by `n`, evaluating

```
(moessner-fold 0 n (lambda (x) 1) (lambda (j x) (- n j)))
```

yields $n!$.

Indeed, inlining the definition of `moessner-fold` here gives rise to the following instance of Moessner's theorem to generate Factorial numbers additively:

$$\sum_{i=0}^{0} \sum_{i=0}^{1} \sum_{i=0}^{2} \sum_{i=0}^{3} \sum_{i=0}^{4} 1 \;=\; 1 \cdot (2 \cdot (3 \cdot (4 \cdot (5 \cdot 1)))) \;=\; 5!$$

In fact, though, any permutation does the job since multiplication is left-permutative [13]:

$$\sum_{i=0}^{3} \sum_{i=0}^{1} \sum_{i=0}^{4} \sum_{i=0}^{0} \sum_{i=0}^{2} 1 \;=\; 4 \cdot (2 \cdot (5 \cdot (1 \cdot (3 \cdot 1)))) \;=\; 5!$$

And Long's first theorem still applies: instead of starting from 1, one can start from another integer and see the resulting Factorial number multiplied by this other integer. Likewise for generalizing the first argument of `moessner-fold` from 0 to a non-negative integer x in Corollary 15:

Corollary 17 (Multiples of factorial numbers)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-fold x n (lambda (x) 1) (lambda (j x) j))
```

yields $n! \cdot (x+1)$.

(Rationale: The stolid instance reads $\sum_{i=0}^{n} \cdots \sum_{i=0}^{1} \sum_{i=0}^{x} 1$ instead of $\sum_{i=0}^{n} \cdots \sum_{i=0}^{1} \sum_{i=0}^{0} 1$.)

7.6 A Generalization: Primitive Iteration

Factorial numbers and integral powers are both specified multiplicatively. So, generalizing, $\forall f : \mathbb{N} \to \mathbb{N}$,

$$\prod_{i=0}^{n} f\,i \;=\; f\,0 \cdot f\,1 \cdot \ldots \cdot f\,n \;=\; f\,0 \cdot (f\,1 \cdot (\ldots \cdot (f\,n \cdot 1) \ldots)) \;=\; \sum_{i=0}^{(f\,0)-1} \cdots \sum_{i=0}^{(f\,n)-1} 1$$

Corollary 18 (Products)**.** For all $f : \mathbb{N} \to \mathbb{N}$ denoted by `f` and for all $n : \mathbb{N}$ denoted by `n`, evaluating

```
(moessner-fold 0 n (lambda (x) 1) (lambda (j x) (- (f j) 1)))
```

yields $\prod_{i=0}^{n} f\,i$.

This corollary suggests a variant of `moessner-fold` where summation starts at 1 instead of at 0, since

$$\sum_{i=0}^{(f\,0)-1} \cdots \sum_{i=0}^{(f\,n)-1} 1 \;=\; \sum_{i=1}^{f\,0} \cdots \sum_{i=1}^{f\,n} 1$$

But be that as it may, we can obtain multiplicative numbers without changing the name of the game:

Corollary 19 (x-fold factorial numbers)**.** For all $n : \mathbb{N}$ denoted by `n` and for all $x : \mathbb{N}$ denoted by `x`, evaluating

```
(moessner-fold x n (lambda (x) 1) (lambda (j _) (* (+ j 1) x)))
```

yields 1 when $x = 0$, the $n+1$st factorial number when $x = 1$, a double factorial number of odd numbers when $x = 2$, the $n+1$st triple factorial number when $x = 3$, the $n+1$st quartic (or 4-fold) factorial number when $x = 4$, the $n+1$st quintuple factorial number when $x = 5$, etc., according to the On-Line Encyclopedia of Integer Sequences.

Inlining the definition of `moessner-fold` here gives rise to the following stolid instance of Moessner's theorem:

$$\sum_{i=0}^{1\cdot x} \sum_{i=0}^{2\cdot x} \sum_{i=0}^{3\cdot x} \cdots \sum_{i=0}^{(n+1)\cdot x} 1$$

(The multiplications in the upper bounds can be obtained by iterated addition, an example of third-order primitive iteration – one over n, one over each upper bound, and one to carry out the multiplication. Ditto for superfactorial numbers.)

7.7 A Corollary About the Additive Generation of Binomial Coefficients

Corollary 20 (Binomial coefficients)**.** For all $x : \mathbb{N}$ denoted by `x` and for all $n : \mathbb{N}$ denoted by `n`, evaluating

```
(moessner-fold x n (lambda (x) 1) (lambda (j x) x))
```

which is Moessner's process without striking-out phase, yields $\binom{x+n}{n}$.

In other words, and remembering Pascal's triangle

$$\begin{array}{ccccccc} & & & \binom{0}{0} & & & \\ & & \binom{1}{0} & & \binom{1}{1} & & \\ & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & \\ \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} \end{array}$$

- for all $x : \mathbb{N}$ denoted by `x`, the function implemented by

  ```
  (lambda (n) (moessner-fold x n (lambda (x) 1) (lambda (j x) x)))
  ```

  is enumerated by the $x+1$st diagonal of Pascal's triangle from right to left (reminder: 0 is the first Peano number, 1 is the second, etc.), and

- for all $n : \mathbb{N}$ denoted by `n`, the function implemented by

  ```
  (lambda (x) (moessner-fold x n (lambda (x) 1) (lambda (j x) x)))
  ```

  is enumerated by the $n+1$st diagonal of Pascal's triangle from left to right (reminder: 0 is the first Peano number, 1 is the second, etc.).

Inlining the definition of `moessner-fold` here gives rise to the following ethereal instances of Moessner's theorem to generate binomial coefficients additively (see Corollary 1 in Section 5):

$$\binom{x+0}{0} = 1 \qquad \binom{x+3}{3} = \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{i_2} 1$$

$$\binom{x+1}{1} = \sum_{i_1=0}^{x} 1 \qquad \binom{x+4}{4} = \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{i_2} \sum_{i_4=0}^{i_3} 1$$

$$\binom{x+2}{2} = \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} 1 \qquad \binom{x+5}{5} = \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} \sum_{i_3=0}^{i_2} \sum_{i_4=0}^{i_3} \sum_{i_5=0}^{i_4} 1$$

7.8 A Corollary About the Additive Generation of Catalan Numbers

In the Online Encyclopedia of Integer Sequences [45], the page for Catalan numbers [39] is presented as the longest of the whole encyclopedia, "for good reasons" since they arise in so many situations – for example, trees, as in Hinze's work [23, Section 5.5], which appears to be the only mention of Catalan numbers in the literature about Moessner's theorem. Here is their inductive definition and a definition in closed form:

$$\begin{cases} C_0 = 1 \\ \forall n : \mathbb{N},\ C_{n+1} = \dfrac{2 \cdot (2 \cdot n + 1) \cdot C_n}{n+2} \end{cases} \qquad \forall n : \mathbb{N},\ C_n = \frac{\binom{2 \cdot n}{n}}{n+1}$$

Corollary 21 (Catalan numbers)**.** For all $n : \mathbb{N}$ denoted by `n`, evaluating

```
(moessner-fold 0 n (lambda (x) 1) (lambda (j x) (+ x 1)))
```

yields the $n+1$st Catalan number.

In other words, the function implemented by the Scheme procedure

```
(lambda (n) (moessner-fold 0 n (lambda (x) 1) (lambda (j x) (+ x 1))))
```

is enumerated by the stream of Catalan numbers.

Inlining the definition of `moessner-fold` here gives rise to the following ethereal instances of Moessner's theorem to generate Catalan numbers additively:

$$C_0 = 1 \qquad C_3 = \textstyle\sum_{i_1=0}^{0} \sum_{i_2=0}^{i_1+1} \sum_{i_3=0}^{i_2+1} 1 \ = \ 5$$

$$C_1 = \textstyle\sum_{i_1=0}^{0} 1 \ = \ 1 \qquad C_4 = \sum_{i_1=0}^{0} \sum_{i_2=0}^{i_1+1} \sum_{i_3=0}^{i_2+1} \sum_{i_4=0}^{i_3+1} 1 \ = \ 14$$

$$C_2 = \textstyle\sum_{i_1=0}^{0} \sum_{i_2=0}^{i_1+1} 1 \ = \ 2 \qquad C_5 = \sum_{i_1=0}^{0} \sum_{i_2=0}^{i_1+1} \sum_{i_3=0}^{i_2+1} \sum_{i_4=0}^{i_3+1} \sum_{i_5=0}^{i_4+1} 1 \ = \ 42$$

At the time of writing, these nested summations are not mentioned in the On-Line Encyclopedia of Integer Sequences. They do, however, correspond to the nested for loops in Example 27.2, page 222, of Ralph Grimaldi's introduction to Fibonacci and Catalan numbers [21], as detailed in Section 9.4.

Long's first theorem applies again: instead of starting from 1, one can start from another integer and see the resulting Catalan number multiplied by this other integer.

In Corollary 21, and as mentioned in Section 5, generalizing the first argument of `moessner-fold` from 0 to a non-negative integer x gives rise to the $x+1$st convolution of Catalan numbers. Generalizing the fourth argument of `moessner-fold` from `(lambda (j x) (+ x 1))` to `(lambda (j x) (+ x i))` where `i` denotes a positive integer, however, does not give rise to a discernible pattern. (Amusingly, though, the consecutive integers 3, 4, 5, and 6 give rise to the consecutive OEIS integer sequences A002293, A002294, A002295, and A002296.) As mentioned in Section 5, letting `i` be `0` gives rise to binomial coefficients, and letting `i` be `(quotient x j)` gives rise to integral powers.

7.9 Moessner's Magic: Primitive Recursion

Could one express Catalan numbers as a stolid instance of Moessner's theorem rather than as the ethereal instance in Section 7.8? This looks unlikely because of the inductive specification of Catalan numbers (stated in Section 7.8). The issue is that $2 \cdot (2 \cdot n + 1) \cdot C_n$ is divisible by $n+2$, but $2 \cdot (2 \cdot n + 1)$ is not. And so even though Catalan numbers are integers, the equality

$$\forall n : \mathbb{N},\ \frac{2 \cdot (2 \cdot n + 1) \cdot C_n}{n+2} \quad = \quad \frac{2 \cdot (2 \cdot n + 1)}{n+2} \cdot C_n$$

is only valid for rational numbers, not for integer division. And since the upper bound of a summation is an integer, not a rational number, this recurrence is not multiplicative and thus not expressible as a stolid instance of Moessner's theorem where the upper bounds are independent of the outer indices. Ditto for the multiplicative definition of Catalan numbers:

$$\forall n : \mathbb{N},\, C_n \;\; = \;\; \prod_{i=2}^{n} \frac{n+i}{i}$$

And therein lies the magic of Moessner's theorem: the upper bounds of the inner sums depend on the indices of the outer summations, and these dependencies account for the transitory rational arithmetic using integer arithmetic in the ethereal instance of Moessner's theorem to express Catalan numbers.

The executive summary and Section 5 presented a series of examples that illustrate these dependencies.

8 Back to Dynamic Programming

For completeness, let us revisit the additive generation of powers and illustrate how to express it using dynamic programming.

Consider, for example, the Scheme procedure that maps x to $(x+1)^5$:

```
(define power-5
  (lambda (x)
    (let* ([p5 (lambda (x)
                 1)]
           [p4 (lambda (x)
                 (Sigma x (lambda (i4)
                            (p5 (quotient (* 6 i4) 5)))))]
           [p3 (lambda (x)
                 (Sigma x (lambda (i3)
                            (p4 (quotient (* 5 i3) 4)))))]
           [p2 (lambda (x)
                 (Sigma x (lambda (i2)
                            (p3 (quotient (* 4 i2) 3)))))]
           [p1 (lambda (x)
                 (Sigma x (lambda (i1)
                            (p2 (quotient (* 3 i1) 2)))))])
      (Sigma x (lambda (i0)
                 (p1 (quotient (* 2 i0) 1)))))))
```

Computing $(x+1)^5$ with `power-5` is achieved by performing $(x+1)^5$ additions. (In general, for any $n : \mathbb{N}$, computing $(x+1)^n$ is achieved by performing $(x+1)^n$ additions.)

Since the local procedures `p5`, `p4`, `p3`, and `p2` are repeatedly called with the same arguments, let us uniformly cache their results into lists, which is the essence of dynamic programming:

```
(define mpower-5
  (lambda (x)
    (let* ([a4 (map (lambda (x)
                      (Sigma x (lambda (i4)
                                 1)))
                    (iotap (* 5 x)))]
           [a3 (map (lambda (x)
                      (Sigma x (lambda (i3)
                                 (list-ref a4 (quotient (* 5 i3) 4)))))
                    (iotap (* 4 x)))]
```

```
          [a2 (map (lambda (x)
                     (Sigma x (lambda (i2)
                                (list-ref a3 (quotient (* 4 i2) 3)))))
                   (iotap (* 3 x)))]
          [a1 (map (lambda (x)
                     (Sigma x (lambda (i1)
                                (list-ref a2 (quotient (* 3 i1) 2)))))
                   (iotap (* 2 x)))])
      (Sigma x (lambda (i0)
                 (list-ref a1 (quotient (* 2 i0) 1)))))))
```

where applying iotap to a non-negative integer n constructs an increasing list from 0 to n. Computing $(x+1)^5$ with mpower-5 is achieved by performing $\frac{5 \cdot 6}{2} \cdot x$ additions. (In general, for any $n : \mathbb{N}$ that is not 2 nor 3, computing $(x+1)^n$ is achieved by performing $x \cdot \sum_{i=0}^{n} i$ additions with dynamic programming – which illustrates the impact of dynamic programming here: from $(x+1)^n$ to $x \cdot \sum_{i=0}^{n} i$ additions.)

We can also fuse the combination of map, Sigma, and iotap into a procedure that is listless:

```
(define fused
  (lambda (x n f)
    (if (<= n 0)
        (errorf 'fused "non-positive input: ~s" n)
        (letrec ([visit (lambda (i j a)
                          (if (= i 0)
                              '()
                              (let ([a (+ (f j) a)])
                                (cons a (visit (- i 1) (+ j 1) a)))))])
          (let ([a (f 0)])
            (cons a (visit (* n x) 1 a)))))))
```

(Such a “loop fusion” is a classical program transformation that dates back to Burge and Landin [5], and a “listless” program is one that does not create intermediate lists [59].) This procedure is listless because the only list it constructs is the result. The corresponding power function reads as follows:

```
(define mpower-5_listless
  (lambda (x)
    (let* ([a4 (fused x 5 (lambda (i4)
                            1))]
           [a3 (fused x 4 (lambda (i3)
                            (list-ref a4 (quotient (* 5 i3) 4))))]
           [a2 (fused x 3 (lambda (i2)
                            (list-ref a3 (quotient (* 4 i2) 3))))]
           [a1 (fused x 2 (lambda (i1)
                            (list-ref a2 (quotient (* 3 i1) 2))))])
      (Sigma x (lambda (i0)
                 (list-ref a1 (quotient (* 2 i0) 1)))))))
```

We can also tune the listless combination so that it does not construct the elements of the lists that will be skipped in the next place:

```
(define fused_
  (lambda (x n f)
    (if (<= n 0)
        (errorf 'fused_ "non-positive input: ~s" n)
        (let ([np (- n 1)])
          (letrec ([visit (lambda (i j a)
                            (if (= i 0)
                                '()
```

```
                              (let ([a (+ (f j) a)])
                                (if (= (remainder j n) np)
                                    (visit (- i 1) (+ j 1) a)
                                    (cons a (visit (- i 1) (+ j 1) a))))))])
            (let ([a (f 0)])
              (cons a (visit (* n x) 1 a))))))))
```

The skipping part is achieved with the inner conditional expression.

The corresponding power function then does not need to skip elements in the lists, making it clearer that $x \cdot \sum_{i=0}^{5} i$ additions are performed:

```
(define mpower-5_listless_
  (lambda (x)
    (let* ([a4 (fused_ x 5 (lambda (i4) 1))]
           [a3 (fused_ x 4 (lambda (i3) (list-ref a4 i3)))]
           [a2 (fused_ x 3 (lambda (i2) (list-ref a3 i2)))]
           [a1 (fused_ x 2 (lambda (i1) (list-ref a2 i1)))])
      (Sigma x (lambda (i0) (list-ref a1 i0))))))
```

Replacing `Sigma x` with `fused_ x 1` makes `mpower-5_listless_` map x to the increasing list $[1^5, 2^5, \cdots, (x+1)^5]$ instead of to $(x+1)^5$, performing the same number of additions thanks to the caching of dynamic programming.

Overall, here is Moessner's process with dynamic programming where the striking-out phase is anticipated:

```
(define moessner_
  (lambda (x n)
    (cond
      [(< n 0)
       (errorf 'moessner_ "negative exponent: ~s" n)]
      [(= n 0)
       1]
      [(= n 1)
       (Sigma x (lambda (i) 1))]
      [else
       (letrec ([visit (lambda (n a)
                         (if (= n 1)
                             (Sigma x (lambda (i)
                                        (list-ref a i)))
                             (visit (- n 1)
                                    (fused_ x n (lambda (i)
                                                  (list-ref a i))))))])
         (visit (- n 1)
                (fused_ x n (lambda (i) 1))))])))
```

In practice, we use arrays rather than lists since we know their size and since unlike lists, they are indexed in constant time, not in linear time.

9 Related Work About Nested Sums

With the exception of Irwin and Lahlou's work described in Section 9.5 and of Baumann's "k-dimensionale Champagnerpyramide" [2], the author could not find much about nested sums in the literature.

9.1 Sums and For Loops

In his lecture notes on Discrete Mathematics [1, Chapter 6], Aspnes connects sums and for loops as an evidence. (To quote, "Mathematicians invented this notation centuries ago because they didn't have for loops.") So for two imperative examples,

- the OCaml expression

  ```
  let a = ref 0
  in for i = 0 to 5 do
       a := !a + 1
     done;
     !a
  ```

  evaluates to $\sum_{i=0}^{5} 1$, i.e., to 6, and

- the OCaml expression

  ```
  let a = ref 0
  in for i = 1 to 5 do
       a := !a + 1
     done;
     !a
  ```

  evaluates to $\sum_{i=1}^{5} 1$, i.e., to 5.

Regarding nested sums, Aspnes mentions that $\sum_{i=1}^{a} \sum_{j=1}^{b} 1$ is a "rather painful" way to multiply a and b, and Example 9, page 408 of the 8th edition of Rosen's textbook about discrete mathematics and its applications [53, Section 6.1.1], illustrates nested for loops to compute the product of n numbers. To wit, the OCaml expression `example_Rosen 2 3 4` evaluates to $\sum_{i_1=1}^{2} \sum_{i_2=1}^{3} \sum_{i_3=1}^{4} 1$, i.e., to $2 \cdot 3 \cdot 4$, i.e., to 24, given the following OCaml declaration:

```
let example_Rosen n1 n2 n3 =
  let k = ref 0
  in for i1 = 1 to n1 do
       for i2 = 1 to n2 do
         for i3 = 1 to n3 do
           k := !k + 1
         done
       done
     done;
     !k
```

In Section 6.1.8 his lecture notes [1], Aspnes also gives $\sum_{i=0}^{n} \sum_{j=0}^{i} (i+1) \cdot (j+1)$ as an example of a nested sum where the upper bound of the inner sum is the index of the outer sum, and mentions that "[f]or larger n, the number of [products] grows quickly."

9.2 Primitive iteration and primitive recursion

A for loop whose body uses its index is said to be primitive recursive and a for loop whose body does not use its index is said to be primitive iterative [13].

9.3 Multiset Coefficients

In Example 6, page 449 of the aforementioned textbook about discrete mathematics and its applications [53, Section 6.5.3], Rosen states the nested for loops that correspond to

$$\sum_{i_1=1}^{x} \sum_{i_2=1}^{i_1} \cdots \sum_{i_n=1}^{i_{n-1}} 1$$

for two given natural numbers x and n (renamed here for notational consistency). He proves combinatorially that the result is the number of combinations of n elements from a multiset with x elements, i.e., from a set where elements can occur several times: the result is the multiset coefficient $\left(\!\binom{x}{n}\!\right)$.

Theorem 3 (Multiset coefficients as nested sums (Rosen))**.** $\forall x \in \mathbb{N}, \forall n \in \mathbb{N}$,

$$\sum_{i_1=1}^{x} \sum_{i_2=1}^{i_1} \cdots \sum_{i_n=1}^{i_{n-1}} 1 \quad = \quad \left(\!\!\binom{x}{n}\!\!\right) \quad = \quad \binom{x+n-1}{n}$$

Proof. By nested first-order induction on n and then on x. □

For comparison, Corollary 1 is proved by induction on n, using Lemma 1 just below. The theorem and its proof rely on the following series of equalities:

$n=0$ $\forall x \in \mathbb{N}, 1 = \binom{x+0}{0}$

$n=1$ $\forall x \in \mathbb{N}, \sum_{i_1=0}^{x} 1 = \sum_{i_1=0}^{x} \binom{i_1+0}{0} = \binom{x+1}{1}$

$n=2$ $\forall x \in \mathbb{N}, \sum_{i_2=0}^{x} \sum_{i_1=0}^{i_2} 1 = \sum_{i_2=0}^{x} \binom{i_2+1}{1} = \binom{x+2}{2}$

$n=3$ $\forall x \in \mathbb{N}, \sum_{i_3=0}^{x} \sum_{i_2=0}^{i_3} \sum_{i_1=0}^{i_2} 1 = \sum_{i_3=0}^{x} \binom{i_3+2}{2} = \binom{x+3}{3}$

etc.

Lemma 1. $\forall x \in \mathbb{N}, \forall n \in \mathbb{N}, \sum_{i=0}^{x} \binom{i+n}{n} = \binom{x+n+1}{n+1}$

Proof. By induction on x. □

Corollary 22. $\forall x \in \mathbb{N}, \forall n \in \mathbb{N}, \sum_{i_1=0}^{x} \sum_{i_2=0}^{i_1} \cdots \sum_{i_n=0}^{i_{n-1}} 1 \ = \ \sum_{i_1=1}^{x+1} \sum_{i_2=1}^{i_1} \cdots \sum_{i_n=1}^{i_{n-1}} 1$

9.4 Catalan Numbers

Catalan numbers (A000108 = [1, 1, 2, 5, 14, 42, 132, 429, 1430, $\cdots$]) can also be obtained using nested for loops (Example 27.2, page 222, of Grimaldi's Introduction to Fibonacci and Catalan numbers [21, Chapter 27] and Example 9, page 150, of the second edition of the Handbook of Discrete and Combinatorial Mathematics [52, Section 3.1.3]) and via a version of Moessner's process without dynamic programming:

$$\forall n \in \mathbb{N}, \sum_{i_1=1}^{1} \sum_{i_2=1}^{i_1+1} \sum_{i_3=1}^{i_2+1} \cdots \sum_{i_n=1}^{i_{n-1}+1} 1 \quad = \quad C_n$$

$$\forall n \in \mathbb{N}, \sum_{i_0=0}^{0} \sum_{i_1=0}^{i_0+1} \sum_{i_2=0}^{i_1+1} \cdots \sum_{i_n=0}^{i_{n-1}+1} 1 \quad = \quad C_{n+1}$$

Grimaldi sketches how the growing indices match the structure of a tree where subtrees have a growing number of subtrees (Figure 27.3, page 222), which justifies how these nested sums compute Catalan numbers, since the number of nodes at level n of this tree is C_n.

In the second equation, and according to the On-Line Encyclopedia of Integer Sequences, generalizing 0 to the natural number x appears to give rise to the $x+1$st convolution of Catalan numbers: A000245 when x is 2, A002057 when x is 3, A000344 when x is 4, A003517 when x is 5, etc.

9.5 Number of Types of Binary Trees of a Given Height

A variant of Moessner's process without dynamic programming gives rise to the following nested sums that compute the number of types of binary trees of height n, i.e., to A002449 $= [1, 1, 2, 6, 26, 166, \cdots[$:

$$\forall n \in \mathbb{N}, \sum_{i_0=0}^{1} \sum_{i_1=0}^{2\cdot i_0+1} \sum_{i_2=0}^{2\cdot i_1+1} \cdots \sum_{i_n=0}^{2\cdot i_{n-1}+1} 1 \quad = \quad \text{A002449}_{n+2}$$

(Replacing the opening 1 with 2 and the 2 factor with 3 gives rise to ternary trees, and likewise for quaternary trees, etc.)

In the web page dedicated to A002449 [44], Irwin conjectures that the following nested sums also compute A002449:

$$\text{A002449}_{n+2} \quad = \quad \sum_{i_1=1}^{2} \sum_{i_2=1}^{2\cdot i_1} \cdots \sum_{i_{n-1}=1}^{2\cdot i_{n-2}} \sum_{i_n=1}^{2\cdot i_{n-1}} 2\cdot i_n, \text{ for } n \geq 1$$

In the material [28] that accompanies A002449 [44], Lahlou proves this conjecture using West's generating trees [60], which is essentially the same coinductive argument as Grimaldi's for Catalan numbers (see Section 9.4). Lahlou also lists a dozen nested summations that compute known integer sequences, including Catalan numbers and Fibonacci numbers:

Theorem 4 (Fibonacci numbers as nested sums (Lahlou))**.** $\forall n \in \mathbb{N}, A_n = F_{n+1}$ where

$$A_n \;=\; \begin{cases} 1 & \textit{if } n=0 \\ 1 & \textit{if } n=1 \\ \sum_{i_2=1}^{1} \sum_{i_3=1}^{3-i_2} \cdots \sum_{i_n=1}^{3-i_{n-1}} (3-i_n) & \textit{otherwise} \end{cases}$$

Proof. By induction on n. □

Irwin's conjecture, West's generating trees, and Lahlou's note are the closest related work here.

Acknowledgments

The author is grateful to Julia Lawall for tirelessly providing grounded comments throughout the writing of this article. Thanks are also due to Kira Kutscher for lending an informed hand with the early articles about Moessner's theorem as well as for pertinent comments, to the NUS library for providing the complete collection of the issues of Sphinx and of Scripta Mathematica, to its librarians for their diligent competence, to Kira Kutscher and Lukas Fesser for their help elucidating Alfred Moessner's particulars, to the anonymous reviewers for their time and comments, and to Annette Bieniusa, Markus Degen, and Stefan Wehr for their editorship.

This article is dedicated to Peter Thiemann in friendly homage for his lifetime of scholarship, academic dedication, and scientific contributions.

References

[1] James Aspnes (2022): *Notes on Discrete Mathematics*. `https://www.cs.yale.edu/homes/aspnes/classes/202/notes.pdf`. 85

[2] Michael Heinrich Baumann (2018): *Die k-dimensionale Champagnerpyramide*. Mathematische Semesterberichte 66, pp. 89–100, doi:10.1007/s00591-018-00236-x. 84

[3] Mark Bickford, Dexter Kozen & Alexandra Silva (2022): *Formalizing Moessner's Theorem and Generalizations in Nuprl*. Journal of Logical and Algebraic Methods in Programming 124, p. 100713, doi:10.1016/j.jlamp.2021.100713. 64, 78

[4] Anders Bondorf & Olivier Danvy (1991): *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types*. Science of Computer Programming 16, pp. 151–195, doi:10.1016/0167-6423(91)90035-V. 77

[5] William H. Burge (1975): *Recursive Programming Techniques*. Addison-Wesley. ISBN 978-0-201-14450-5. 83

[6] Rod M. Burstall & John Darlington (1977): *A Transformational System for Developing Recursive Programs*. Journal of the ACM 24(1), pp. 44–67, doi:10.1145/321992.321996. 61

[7] Alonzo Church (1941): *The Calculi of Lambda-Conversion*. Princeton University Press. ISBN 978-0-691-08394-0. 59, 77

[8] Christian Clausen, Olivier Danvy & Moe Masuko (2014): *A Characterization of Moessner's Sieve*. Theoretical Computer Science 546, pp. 244–256, doi:10.1016/J.TCS.2014.03.012. 62, 64, 69

[9] Charles Consel & Olivier Danvy (1993): *Tutorial Notes on Partial Evaluation*. In Susan L. Graham, editor: Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, ACM Press, Charleston, South Carolina, pp. 493–501, doi:10.1145/158511.158707. 77

[10] John H. Conway & Richard K. Guy (1996): *Moessner's Magic*. In: The Book of Numbers, Springer, pp. 63–65, doi:10.1007/978-1-4612-4072-3. 57, 64, 78

[11] John H. Conway & Richard K. Guy (1996): *The Polygonal Numbers*. In: The Book of Numbers, Springer, pp. 38–42, doi:10.1007/978-1-4612-4072-3. 91

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2001): *Introduction to Algorithms*, second edition. The MIT Press, Cambridge, Massachusetts. ISBN 978-0-262-04630-5. 60

[13] Olivier Danvy (2023): *Folding Left and Right Matters: Direct Style, Accumulators, and Continuations*. Journal of Functional Programming 33(e2), doi:10.1017/S0956796822000156. 73, 79, 85

[14] Olivier Danvy (2024): *Nested Summations*. In Mayur Naik, Fernando Magno Quintão Pereira & Ben L. Titzer, editors: Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday (JENSFEST 2024), Pasadena, California, doi:10.1145/3694848.3694858. ACM 979-8-4007-1257-9/24/10. 72

[15] Elena Deza & Michel Marie Deza (2012): *Figurate Numbers*. World Scientific, doi:10.1142/8188. 91

[16] Leonard Eugene Dixon (1920): *History of the Theory of Numbers, Vol. II: Diophantine Analysis*. Carnegie Institution of Washington. 91

[17] R. Kent Dybvig (1996): *The Scheme Programming Language*, second edition. Prentice Hall. ISBN 978-0-262-51298-5. 59

[18] Daniel P. Friedman & David S. Wise (1976): *CONS Should Not Evaluate its Arguments*. In S. Michaelson & Robin Milner, editors: Third International Colloquium on Automata, Languages, and Programming, Edinburgh University Press, Edinburgh, Scotland, pp. 257–284. 60

[19] Albert Gloden (1947): *Moessner Triplets*. Scripta Mathematica 13, p. 58. 63

[20] Ronald L. Graham, Donald E. Knuth & Oren Patashnik (1994): *Concrete Mathematics*, 2nd edition. Addison-Wesley. ISBN 0-201-55802-5. 59

[21] Ralph P. Grimaldi (2012): *Fibonacci and Catalan Numbers – An Introduction*. John Wiley and Sons, doi:10.1002/9781118159743. ISBN 978-0-470-63157-7. 81, 86

[22] Ralf Hinze (2008): *Scans and Convolutions – A Calculational Proof of Moessner's Theorem*. In Sven-Bodo Scholz & Olaf Chitil, editors: *Implementation and Application of Functional Languages, 20th International Workshop, IFL 2008*, *Lecture Notes in Computer Science* 5836, Springer, Hatfield, UK, pp. 1–24, doi:10.1007/978-3-642-24452-0_1. 64, 78

[23] Ralf Hinze (2011): *Concrete Stream Calculus – An Extended Study*. *Journal of Functional Programming* 20(5-6), pp. 463–535, doi:10.1017/S0956796810000213. 64, 78, 81

[24] Ross Honsberger (1991): *More Mathematical Morsels*. *The Dolciani Mathematical Exposition* 10, The Mathematical Association of America, doi:10.1090/dol/010. ISBN 978-1-4704-5838-6. 63, 64

[25] Stephen C. Kleene (1952): *Introduction to Metamathematics*. Bibliotheca Mathematica, North-Holland Publishing Co., Amsterdam, The Netherlands. ISBN 978-0-7204-2103-3. 77

[26] Dexter Kozen & Alexandra Silva (2013): *On Moessner's Theorem*. *The American Mathematical Monthly* 120(2), pp. 131–139, doi:10.4169/amer.math.monthly.120.02.131. 64, 78

[27] Robbert Krebbers, Louis Parlant & Alexandra Silva (2016): *Moessner's Theorem: An Exercise in Coinductive Reasoning in Coq*. In Erika Ábrahám, Marcello M. Bonsangue & Einar Broch Johnsen, editors: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 9660, Springer, pp. 309–324, doi:10.1007/978-3-319-30734-3_21. 64

[28] Chams Lahlou (2019): *A formula for some integer sequences that can be described by generating trees*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A002449/a002449.pdf`. 87

[29] Calvin T. Long (1966): *On the Moessner Theorem on Integral Powers*. *The American Mathematical Monthly* 73(8), pp. 846–851, doi:10.1080/00029890.1966.11970851. 62, 64, 75, 78

[30] Calvin T. Long (1982): *Mathematical Excitement—The Most Effective Motivation*. *The Mathematics Teacher* 75(5), pp. 413–415, doi:10.5951/MT.75.5.0413. 64

[31] Calvin T. Long (1982): *Strike It Out—Add It Up*. *The Mathematical Gazette* 66(428), pp. 273–277, doi:10.2307/3615513. 64, 66

[32] Calvin T. Long (1986): *A Note On Moessner's Process*. *The Fibonacci Quarterly* 24(4), pp. 349–355, doi:10.1080/00150517.1986.12429746. 64, 76, 78

[33] Alfred Moessner (1941): *Triplets Again*. *Scripta Mathematica* 8, p. 14. 63

[34] Alfred Moessner (1947): *Curiosa 161: A Curious Magic Square*. *Scripta Mathematica* 13, p. 234. 63

[35] Alfred Moessner (1951): *Eine Bemerkung über die Potenzen der natürlichen Zahlen*. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematischnaturwissenschaftliche Klasse* 29(3), p. 9. 58, 62, 63

[36] Alfred Moessner (1952): *Curiosa 306: All-Prime Magic Squares*. *Scripta Mathematica* 18(3–4), p. 303. 63

[37] Alfred Moessner (2024): *Mößner, Alfred*. zbMATH Open. Available at `https://zbmath.org/authors/mossner.alfred`. 63

[38] Milad Niqui & Jan Rutten (2011): *A Proof of Moessner's Theorem by Coinduction*. *Higher-Order and Symbolic Computation* 24(3), pp. 191–206, doi:10.1007/S10990-012-9082-7. 64

[39] OEIS Foundation Inc. (2023): *Entry A000108 (the stream of Catalan numbers)*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A000108`. 81

[40] OEIS Foundation Inc. (2023): *Entry A000142 (the stream of Factorial numbers)*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A000142`. 78

[41] OEIS Foundation Inc. (2023): *Entry A000270 (the stream of triangular numbers)*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A000270`. 92

[42] OEIS Foundation Inc. (2023): *Entry A000326 (the stream of pentagonal numbers)*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A000326`. 91

[43] OEIS Foundation Inc. (2023): *Entry A000384 (the stream of hexagonal numbers)*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A000384`. 92

[44] OEIS Foundation Inc. (2023): *Entry A002449*. In: *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/A002449`. 87

[45] OEIS Foundation Inc. (2023): *The On-Line Encyclopedia of Integer Sequences*. `https://oeis.org/`. 64, 72, 81

[46] Ivan Paasche (1952): *Ein neuer Beweis des Moessnerschen Satzes*. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematischnaturwissenschaftliche Klasse* 30(1), pp. 1–5. 62, 63

[47] Ivan Paasche (1953/54): *Ein zahlentheoretische-logarithmischer 'Rechenstab'. Zum Satz von Moessner*. *Der Mathematische und Naturwissenschaftliche Unterricht* 6, pp. 26–28. 58, 63, 64, 78

[48] Ivan Paasche (1954-1956): *Eine Verallgemeinerung des Moessnerschen Satzes*. *Compositio Mathematica* 12, pp. 263–270. 64

[49] Ivan Paasche (1955): *Beweis des Moessnerschen Satzes mittels linearer Transformationen*. *Archiv der Mathematik* 6, pp. 194–199, doi:10.1007/BF01900739. 63

[50] Oskar Perron (1951): *Beweis des Moessnerschen Satzes*. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematischnaturwissenschaftliche Klasse* 29(4), pp. 31–34. 63

[51] Burkard Polster (2021): *The Moessner Miracle. Why wasn't this discovered for over 2000 years?* Mathologer (short video documentary). `https://www.youtube.com/watch?v=rGlpyFHfMgI&t=235s&ab_channel=Mathologer`. 59

[52] Kenneth H. Rosen, editor (2018): *Handbook of Discrete and Combinatorial Mathematics*, second edition. Chapman & Hall, doi:10.1201/9781315156484. 86

[53] Kenneth H. Rosen (2019): *Discrete Mathematics and its Applications*, eighth edition. McGraw Hill. ISBN 9781584887805. 59, 85, 86

[54] Hans Salié (1952): *Bemerkung zu einem Satz von A. Moessner*. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematischnaturwissenschaftliche Klasse* 30(2), pp. 7–11. 63

[55] Saed Samadi, M. Omair Ahmad & M. N. Shanmukha Swamy (2005): *Multiplier-free structures for exact generation of natural powers of integers*. In: *International Symposium on Circuits and Systems (ISCAS 2005)*, IEEE, Kobe, Japan, pp. 1146–1149, doi:10.1109/ISCAS.2005.1464796. 58, 59

[56] John G. Slater (1983): *Strike It Out—Some Exercises*. *The Mathematical Gazette* 67(442), pp. 288–290, doi:10.2307/3617270. 64, 66

[57] Uladzimir Treihis (2024): *Formalizing the Original Proofs of Moessner's Theorem in the Coq Proof Assistant*. BSc thesis, Yale-NUS College, Singapore. 64

[58] Peter Urbak (2017): *A Formal Study of Moessner's Sieve*. MSc thesis, Department of Computer Science, Aarhus University, Arhus, Denmark, doi:10.7146/aul.213.154. `https://ebooks.au.dk/aul/catalog/book/213`. 64

[59] Philip Wadler (1984): *Listlessness is better than laziness*. In Guy L. Steele Jr., editor: *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM Press, Austin, Texas, pp. 282–305, doi:10.1145/800055.802020. 83

[60] Julian West (1996): *Generating Trees and Forbidden Subsequences*. *Discrete Mathematics* 157, pp. 363–374, doi:10.1016/S0012-365X(96)83023-8. 87

[61] Jan van Yzeren (1959): *A Note on an Additive Property of Natural Numbers*. *The American Mathematical Monthly* 66(1), pp. 53–54, doi:10.2307/2309925. 63, 64

A Content of the accompanying .scm file

The accompanying `.scm` file contains an implementation in Scheme of the whole article together with a comprehensive collection of tests.

B Content of the accompanying .v file

The accompanying `.v` file contains a formalization in Coq of part of the executive summary, Section 5, Section 6, and Section 9.

C A Curiosa About Polygonal Numbers

C.1 Introduction

Polygonal numbers are classical, textbook material about the positive integers whose successor constructors in base 1 (Peano numbers) can be laid down on the plane in a way that represents a polygon: a triangle (the numbers are triangular: 1, 3, 6, 10, 15, ...), a square (the numbers are square: 1, 4, 9, 16, 25, ...), a pentagon (the numbers are pentagonal: 1, 5, 12, 22, 35, ...), etc. They are known since the Greeks [16] and are a thoroughly explored topic with many beautiful properties [11,15]. For all positive natural numbers n, $\frac{n\cdot(n+1)}{2}$ denotes a triangular number, n^2 denotes a square number, $\frac{n\cdot(3\cdot n-1)}{2}$ denotes a pentagonal number, $2\cdot n^2 - n$ denotes an hexagonal number, and more generally, $\sum_{i=0}^{n}(k\cdot i+1)$ denotes a $k+2$nd-order polygonal number, which can also be expressed using triangular numbers of order n, for all positive natural numbers k.

C.2 The Curiosa

In the course of Section 3.4, page 67, the following sum pops up (where n is i_1, i is i_2, and i_1 is quantified in a sum, not with λ):

$$\lambda n.\sum_{i=0}^{2\cdot n}\frac{i}{2} : \mathbb{N}\to\mathbb{N}$$

As it happens, this function is enumerated by the stream of square numbers, i.e., $[0, 1, 4, 9, 16, \cdots[$. So it is equivalent to $\lambda n.n^2$. In words, summing twice as many successive halves yields a square number:

Theorem 5 (Square numbers as bounded summations of integer quotients)**.**
$\forall n \in \mathbb{N}, \sum_{i=0}^{2\cdot n}\lfloor\frac{i}{2}\rfloor = n^2$

Proof. By induction on n.
Base case ($n=0$): We want to show $\sum_{i=0}^{2\cdot 0}\lfloor\frac{i}{2}\rfloor = 0^2$.
$\sum_{i=0}^{2\cdot 0}\lfloor\frac{i}{2}\rfloor = \sum_{i=0}^{0}\lfloor\frac{i}{2}\rfloor = \lfloor\frac{0}{2}\rfloor = 0 = 0^2$
Induction step ($n = n'+1$): Assuming $\sum_{i=0}^{2\cdot n'}\lfloor\frac{i}{2}\rfloor = n'^2$, we want to show $\sum_{i=0}^{2\cdot(n'+1)}\lfloor\frac{i}{2}\rfloor = (n'+1)^2$.
$\sum_{i=0}^{2\cdot(n'+1)}\lfloor\frac{i}{2}\rfloor = \sum_{i=0}^{2\cdot n'+2}\lfloor\frac{i}{2}\rfloor = \sum_{i=0}^{2\cdot n'}\lfloor\frac{i}{2}\rfloor + \sum_{i=2\cdot n'+1}^{2\cdot n'+2}\lfloor\frac{i}{2}\rfloor = n'^2 + \lfloor\frac{2\cdot n'+1}{2}\rfloor + \lfloor\frac{2\cdot n'+2}{2}\rfloor = n'^2 + n' + (n'+1) = n'^2 + 2\cdot n' + 1 = (n'+1)^2$ □

Curiosity compels one to look at

$$\lambda n.\sum_{i=0}^{3\cdot n}\frac{i}{3} : \mathbb{N}\to\mathbb{N}$$

which – via the On-Line Encyclopedia of Integer Sequences – is enumerated by the stream of pentagonal numbers [42], i.e., $[0, 1, 5, 12, 22, 35, \cdots[$. So it is equivalent to $\lambda n.\frac{n\cdot(3\cdot n-1)}{2}$. In words, summing three times as many successive thirds yields a pentagonal number:

Theorem 6 (Pentagonal numbers as bounded summations of integer quotients)**.**
$\forall n \in \mathbb{N}, \sum_{i=0}^{3\cdot n}\lfloor\frac{i}{3}\rfloor = \frac{n\cdot(3\cdot n-1)}{2}$

As for
$$\lambda n.\sum_{i=0}^{4\cdot n}\frac{i}{4} : \mathbb{N} \to \mathbb{N},$$
it is enumerated by the stream of hexagonal numbers [43], i.e., $[0, 1, 6, 15, 28, 45, \cdots[$. So it is equivalent to $\lambda n.2\cdot n^2 - n$. In words, summing four times as many successive quarters yields an hexagonal number:

Theorem 7 (Hexagonal numbers as bounded summations of integer quotients)**.**
$\forall n \in \mathbb{N}, \sum_{i=0}^{4\cdot n}\lfloor\frac{i}{4}\rfloor = 2\cdot n^2 - n$

And since
$$\lambda n.\sum_{i=0}^{1\cdot n}\frac{i}{1} = \lambda n.\sum_{i=0}^{n} i : \mathbb{N} \to \mathbb{N}$$
is enumerated by the stream of triangular numbers [41], i.e., $[0, 1, 3, 6, 10, 15, \cdots[$, it is equivalent to $\lambda n.\frac{n\cdot(n+1)}{2}$. In words, summing the first successive natural numbers yields a triangular number:

Theorem 8 (Triangular numbers as bounded summations of natural numbers)**.**
$\forall n : \mathbb{N}, \sum_{i=0}^{1\cdot n}\lfloor\frac{i}{1}\rfloor = \frac{n\cdot(n+1)}{2}$

These observations lead one to the following theorem:

Theorem 9 (Polygonal numbers as bounded summations of increasing quotients (June 2023))**.** For any given positive integer k and for all natural numbers n, $\sum_{i=0}^{k\cdot(n+1)}\lfloor\frac{i}{k}\rfloor$ is a $k+2$nd polygonal number. Put otherwise, $\sum_{i=0}^{k\cdot(n+1)}\lfloor\frac{i}{k}\rfloor = \sum_{i=0}^{n}(k\cdot i+1) = k\cdot\sum_{i=0}^{n} i + n + 1$ in traditional summative form and in terms of triangular numbers.

To prove the first equality, the following pair of identities comes handy: $\forall x : \mathbb{N}, \forall y : \mathbb{N}, \forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$,
$$\sum_{i=0}^{(x+1)\cdot(y+1)} f\,i = \sum_{i=0}^{x\cdot(y+1)+y} f\,i \;+\; f\,((x+1)\cdot(y+1)) \quad \wedge \quad \sum_{i=0}^{x\cdot(y+1)+y} f\,i = \sum_{i=0}^{x}\sum_{j=0}^{y} f\,(i\cdot(y+1)+j)$$

(To start the proof, observe that since k is a positive natural number, there exists a natural number k' such that $k = k'+1$, and then consider $\sum_{i=0}^{(n+1)\cdot(k'+1)}\lfloor\frac{i}{k'+1}\rfloor$. Extra hint: $\sum_{j=0}^{k'}\lfloor\frac{i\cdot(k'+1)+j}{k'+1}\rfloor = i$.)

In words, polygonal numbers can be characterized as bounded summations of increasing quotients of the polygonal order, minus 2. In each sum, k occurs both in the upper bound of the summation as a multiplier and in its body as a divisor.

Now what does this characterization buy us? In the author's admittedly limited experience, not terribly much: It is simple to formalize in Coq and it is well suited to reason, e.g., about the difference between two polygonal numbers, but otherwise it does not appear to provide an insightful new edge to prove other properties, nor an ingenious new edge to compute polygonal numbers more efficiently.

C.3 Conclusion

Characterizing polygonal numbers as sums of increasing quotients (twice as many successive halves, three times as many successive thirds, etc.) is aligned with the additive concerns of the Greeks, thought-provokingly simple, and pleasingly uniform – perhaps because its nature is computational. Still, this characterization does not propel number theory forward nor does it provide a more efficient way of computing polygonal numbers. At any rate, for all its simplicity, this characterization is new, which is unexpected considering that polygonal numbers date back to Pythagoras. As such, it makes a fun Curiosa in memory of Alfred Moessner.