Theoretical Computer Science 546 (2014) 244–256

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A characterization of Moessner's sieve

Christian Clausen [a], Olivier Danvy [a,*], Moe Masuko [b]

[a] *Department of Computer Science, Aarhus University, Denmark*
[b] *Ochanomizu University, Japan*

ARTICLE INFO

Available online 6 March 2014

Keywords:
Moessner's sieve
Coq

ABSTRACT

Given a positive natural number n, Moessner's sieve constructs the stream of positive natural numbers exponentiated at that rank: 1^n, 2^n, 3^n, etc., without performing any multiplications. Moessner's sieve starts from the stream of positive natural numbers, and proceeds iteratively: first it strikes out every nth number and maps the result into a stream of partial sums; then it strikes out every $(n-1)$th number and maps the result into a stream of partial sums; [...]; then it strikes out every 3rd number and maps the result into a stream of partial sums; and then it strikes out every 2nd number and maps the result into a stream of partial sums. Moessner's theorem states that the end result of this sieve is the stream of positive natural numbers exponentiated at rank n. In this article, we generalize Moessner's sieve to rank 0, we formalize it with the Coq proof assistant, we present generating functions for the numbers that are struck out at each iteration, we define its left inverse, and we touch upon Long's theorem, which generalizes Moessner's theorem.

Given a natural number n, we characterize the successive streams of numbers that are struck out as enumerating the successive monomials of the binomial expansion of $(1+x)^n$. To this end, we initialize Moessner's sieve with a stream that starts with 1 and continues with 0's, and we add a final step where we pick every number in the resulting stream. Since the last monomial is x^n, Moessner's theorem at rank n follows as a corollary.

1. Introduction

A child's first adventure with the Queen of Mathematics can begin as soon as (s)he can add and multiply natural numbers, distinguish between odd and even numbers, and be aware of square numbers:

Once upon a time, there was an odd little adder who liked to add numbers. At school, he had learned about odd and even numbers. Being an odd little adder, he got into his head to add as many odd numbers as he could. So starting from 1, the first odd number, he added to it the second odd number: $1+3=4$. And then he added to 4 the third odd number: $4+5=9$. And then he added to 9 the fourth odd number: $9+7=16$. And then he added to 16 the fifth odd number: $16+9=25$.

Then he paused, contemplating his results. There was something funny about them... especially if he initially started from 0 and added to it the first odd number: $0+1=1$. So he added to 25 the next odd number: $25+11=36$, and yes, the result was also a square!

[*] Corresponding author.
E-mail address: danvy@cs.au.dk (O. Danvy).

http://dx.doi.org/10.1016/j.tcs.2014.03.012
0304-3975/© 2014 Elsevier B.V. All rights reserved.

So he quickly added more and more odd numbers, and every time, the result was a square—to be precise: the next square in the list of successive squares: 1, 4, 9, 16, 25, 36, 49, 64, etc.

And he wondered: Will I always obtain squares if I keep adding odd numbers?

With a modicum of pedagogy, the story of the odd little adder can pave the way to mathematical induction in the mind of a child.

Sixty years ago [11], Alfred Moessner discovered how to obtain not only the successive squares, but also the successive cubes, etc. In effect, he devised a sieve that starts from the list of positive natural numbers, and operates as follows. Given a natural number $n > 1$, iteratively

- elide every nth number and form a resulting list of partial sums;
- elide every $(n-1)$th number and form a resulting list of partial sums;
- ...
- elide every 3rd number and form a resulting list of partial sums; and
- elide every 2nd number and form a resulting list of partial sums.

Moessner's theorem states that the end result of the sieve is the list of successive exponents of n: 1^n, 2^n, 3^n, etc. When $n = 2$, there is only one iteration: eliding every second element in the list of positive natural numbers gives the list of odd numbers, and forming a resulting list of partial sums yields the list of successive squares, as in the story of the odd little adder.

1.1. Related work

Moessner's theorem was first stated as a conjecture [11]. It was then proved using mathematical induction by Oskar Perron [14], and then obtained as a corollary of a more general theorem about generating functions by Ivan Paasche [13] and as an application of Horner's algorithm to a polynomial of degree n, for any given n, by Jan van Yzeren [20]. Moessner's sieve iteratively constructs rows of successive numbers.[1] Together, these successive rows offer a triangular appearance as more and more numbers are struck out. Van Yzeren was the first to view the diagonals of these triangles as (binomial) coefficients of polynomials [20].

Moessner's sieve has been applied to an arbitrary list by Hans Salié [15] instead of to the list of positive natural numbers, and to a list following an arithmetic progression by Calvin Long [6], using a generalization of Pascal's triangle. It has been used by Long as a motivational device for students in high school [7,8,18]. Subsequently, Long has documented his findings about tuning the initial list and the period of elision [9]. In particular, he has related the initial list of elided numbers with the result of Moessner's sieve, pointing out that when the period increases regularly, the sieve maps sums and products into products and exponentiations. This way of obtaining powers without performing multiplications makes Moessner's sieve relevant for signal processing [16], showing that it is not merely a number-theoretical curiosity but that it actually contributes to the unreasonable effectiveness of mathematics in computing.

Five years ago [3], Ralf Hinze put Moessner's theorem into a programming-language perspective, presenting an elegant calculational proof for it and then illustrating it in an encyclopedic article about streams [4]. Milad Niqui and Jan Rutten's take on Moessner's theorem is a proof pearl of co-induction [12]. Dexter Kozen and Alexandra Silva presented an algebraic proof for a generalization of Moessner's theorem [5], and together with Mark Bickford, they were the first to formalize Moessner's theorem in a proof assistant, Nuprl, as a corollary of their generalization [2].

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 an illustration for a test bed. Accordingly, the present article reflects what is in our mind as computer scientists and functional programmers: we want to program Moessner's sieve to understand not just how it works but also why it works.

1.2. This work

To understand why Moessner's theorem holds in terms of functional programming, we formalized Moessner's sieve using streams (lazy lists) of natural numbers in the Coq proof assistant [1]. We implemented partial sums with a function traversing a stream with an accumulator holding the sum computed so far in the stream (Section 2.2). That led us to writing 'master lemmas' quantifying over this accumulator. We then realized that at each rank n, the successive master lemmas involved the successive monomials occurring in the binomial expansion of $(1+x)^n$, and that these monomials are the generating functions of the numbers that are elided at each iteration of Moessner's sieve. Therefore van Yzeren's view of diagonals as polynomial coefficients in the successive rows produced by Moessner's sieve [20] and Hinze's 'binomial process' [3, Section 6] are our closest related work.

To completely match the number of streams, the number of monomials, and the number of iterations of Moessner's sieve, we have adjusted it as follows:

[1] We are putting forward "sieve" as a noun in reference to the sieve of Eratosthenes, to replace the less descriptive nouns "process," "procedure," etc.

- rather than starting from the stream of positive natural numbers, it is tempting to start from the constant stream of ones, like Niqui and Rutten [12], and as first suggested by Long [8];
- however, the stream of ones is itself a stream of partial sums of the stream that starts with 1 and continues with 0's; starting from that stream is nicer in several ways, e.g., to define the left inverse of Moessner's sieve (Section 4) and to very modestly restate Long's theorem (Section 5).

As a byproduct, Moessner's sieve is no longer "just" defined to yield squares of natural numbers at rank 2, cubes at rank 3, etc.—it is also defined at rank 0 (and yields 1^0, 2^0, 3^0, etc.) and at rank 1 (and yields 1^1, 2^1, 3^1, etc.).

For any given natural number n, we form the $n+1$ streams of the numbers that are successively elided in the course of Moessner's sieve, given a stream that starts with 1 and continues with 0's: every $(n+1)$th number, every nth number, every $(n-1)$th number, …, every 3rd number, every 2nd number, and finally every number. It is our observation—and the enabling contribution of this article—that the generating functions of these numbers are the successive monomials of the binomial expansion of $(1+x)^n$:

Rank $n=0$: $\lambda x.1$
  since $(1+x)^0=1$.

Rank $n=1$: $\lambda x.1$, $\lambda x.x$,
  since $(1+x)^1=1+x$.

Rank $n=2$: $\lambda x.1$, $\lambda x.2x$, and $\lambda x.x^2$
  since $(1+x)^2=1+2x+x^2$.

Rank $n=3$: $\lambda x.1$, $\lambda x.3x$, $\lambda x.3x^2$, and $\lambda x.x^3$
  since $(1+x)^3=1+3x+3x^2+x^3$.

Rank $n=4$: $\lambda x.1$, $\lambda x.4x$, $\lambda x.6x^2$, $\lambda x.4x^3$, and $\lambda x.x^4$
  since $(1+x)^4=1+4x+6x^2+4x^3+x^4$.

etc.

So for example, at rank 3, the third iteration of Moessner's sieve strikes out the numbers 3, 12, 27, 48, 75, …, which are the results of applying the function $\lambda x.3x^2$ to 1, 2, 3, 4, 5, … .

For each rank n, the final generating function captures Moessner's theorem at that rank: the remaining numbers in the stream are the result of applying the function $\lambda x.x^n$ to 1, 2, 3, 4, 5, … .

1.3. Overview

The rest of this article is structured as follows. Section 2 presents our domain of discourse, i.e., how to formalize streams and reason about them in Coq, all the way to the story of the odd little adder (Section 2.3). Section 3 formalizes Moessner's sieve for any given rank, and characterizes the numbers it elides as the enumeration of the successive monomials in the binomial expansion of $1+x$ at that rank, for $x=1,2,3,\ldots$. Section 4 introduces a left inverse for Moessner's sieve. Section 5 addresses Long's theorem, which generalizes Moessner's theorem, and Section 6 concludes.

Prerequisites and notations: We assume a basic familiarity with mathematical induction as can be gathered from, well, yes, Glynn Winskel's textbook [19]. We only use Coq at an elementary level [1].

Coq offers a built-in inductive type for natural numbers,

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.
```

and of its rendering of natural numbers as `0` for `O`, `1` for `S O`, `2` for `S (S O)`, `3` for `S (S (S O))`, etc. Inductive data are incrementally deconstructed by pattern matching, as in ML.

Computing over inductive data such as natural numbers is naturally carried out with structurally recursive functions. For example, here is a structurally recursive function (declared with the keyword `Fixpoint`) to compute a binomial coefficient:

```
Fixpoint make_binomial_coefficient (n k : nat) : nat :=
  match k with
    | 0 => 1
    | S k' => match n with
                | 0 => 0
                | S n' => make_binomial_coefficient n' k' +
                          make_binomial_coefficient n' k
              end
  end.
```

Note how the two natural numbers are deconstructed by structural cases, and how each recursive call is over structurally smaller data.

Coq also offers a built-in coinductive type for streams (i.e., lazy lists) of natural numbers and its constructor `Cons` [1, Chapter 13]:

```
CoInductive Stream_nat : Type :=
| Cons : nat -> Stream_nat -> Stream_nat.
```

Given a natural number and a stream of natural numbers, `Cons` yields a new stream that starts with this given natural number and continues with this given stream of natural numbers. Coinductive data are incrementally deconstructed by pattern matching, as in ML. For example, the following function maps a natural number *m* and a stream of natural numbers into a list of length *m* representing the prefix of the stream.

```
Fixpoint prefix (m : nat) (ns : Stream nat) : list nat :=
  match m with
    | 0 => nil
    | S m' => match ns with
                | Cons n ns' => n :: (prefix m' ns')
              end
  end.
```

In Coq, the two list constructors "nil" and "cons" are represented as `nil` and `::`, which is infix. This function comes handy to peek into a stream.

We make use of the fact that 0 is the first natural number, 1 the second, 2 the third, etc., to index a stream. So, for example, regarding the function `drop` that periodically drop numbers out from its input stream, we will write "`drop 2 ns`" to drop every third element from the stream `ns` since 2 is the third natural number.

Notationally, "`x ^ n`" denotes the exponentiation of the natural number denoted by `x` with the natural number denoted by `n`, so that `2 ^ 10` evaluates to `1024` (i.e., to 1024 applications of the constructor `S` to `O`).

Finally, we also make use of the tactic `unfold` that replaces a name declared in a definition by what it stands for (i.e., by its definiens).

2. The Coq domain of discourse

We first illustrate how to reason about streams of natural numbers (Section 2.1). We then formalize two functions: one maps a stream of natural numbers into the stream of its partial sums (Section 2.2), and the other periodically drops elements from a given stream of natural numbers (Section 2.4). This formalization puts us in position to formalize the odd little adder's discovery (Section 2.3). We also need a function to periodically pick elements from a given stream of natural numbers (Section 2.5).

2.1. Streams and reasoning about them

For a first example, let us define the constant stream of the natural number 0, named `zeroes` with a corecursive function (declared with the keyword `CoFixpoint`):

```
CoFixpoint zeroes : Stream nat := Cons 0 zeroes.
```

The identifier `zeroes` denotes the coinductive construction

```
Cons 0 (Cons 0 (Cons 0 ...)).
```

An expressive constraint of Coq is that any occurrence of something defined with `CoFixpoint`, on the right-hand side, must be "guarded" by a constructor. It is the case here: on the right-hand side, `zeroes` occurs as the second operand of `Cons`.

More generally, let us define a generic function to construct streams, named `make_Stream_nat`:

```
CoFixpoint make_Stream_nat (f : nat -> nat) (n : nat) : Stream nat :=
  Cons n (make_Stream_nat f (f n)).
```

(Note how the occurrence of `make_Stream_nat`, on the right-hand side, is guarded by `Cons`.) Applying `make_Stream_nat` to a function `f` and to an initial natural number `m` yields the coinductive construction

```
Cons m (Cons (f m) (Cons (f (f m)) (Cons (f (f (f m))) ...))).
```

We can also define a generic map function over streams of natural numbers:

```
CoFixpoint map_Stream_nat (f : nat -> nat) (ns : Stream nat) : Stream nat :=
  match ns with
    | Cons n ns' => Cons (f n) (map_Stream_nat f ns')
  end.
```

To reason about streams, we use the following deceivingly simple definition and theorem, which are due to Christine Paulin-Mohring [1, Chapter 13]:

```
Definition decompose_Stream_nat (ns : Stream nat) : Stream nat :=
  match ns with
    | Cons n ns' => Cons n ns'
  end.


Theorem decomposition_Stream_nat :
  forall ns : Stream nat,
    ns = decompose_Stream_nat ns.
```

For example, we can lay some ground to reason equationally about streams with the following unfolding lemma:

```
Lemma unfold_zeroes :
  zeroes = Cons 0 zeroes.
```

To prove it, we use the theorem above as a rewriting rule on the first occurrence of `zeroes`, we inline the definition of `decompose_Stream_nat`, replacing its name by its definiens, and in the pioneering steps of Manna and Waldinger [10], we unfold and then fold the definition of `zeroes`. The resulting goal is a Leibniz equality, which holds by reflexivity:

```
Proof.
  rewrite -> (decomposition_Stream_nat zeroes) at 1.
  unfold decompose_Stream_nat.
  unfold zeroes at 1; fold zeroes.
  reflexivity.
Qed.
```

Along the same lines, we can lay a similar ground to reason equationally about streams constructed by `make_Stream_nat` and `map_Stream_nat`:

```
Lemma unfold_make_Stream_nat :
  forall (f : nat -> nat) (n : nat),
    make_Stream_nat f n = Cons n (make_Stream_nat f (f n)).


Lemma unfold_map_Stream_nat :
  forall (f : nat -> nat) (n : nat) (ns' : Stream nat),
    map_Stream_nat f (Cons n ns') = Cons (f n) (map_Stream_nat f ns').
```

In general, however, one cannot always reason equationally about streams using Leibniz equality. For example, we cannot traverse two streams recursively to test whether they are equivalent—this traversal would not terminate. Indeed, traversing a data structure recursively (using a function defined with the `Fixpoint` keyword in Coq) only makes sense if this data structure was constructed inductively (and defined with the `Inductive` keyword in Coq). Streams, however, are constructed coinductively (and defined with the `CoInductive` keyword in Coq), and are traversed corecursively (using a function defined with the `CoFixpoint` keyword in Coq). They are equivalent if they coinductively construct the same stream, i.e., if they are *bisimilar* [17]. Here is a definition of bisimilarity between two streams of natural numbers in Coq:

```
CoInductive bisimilar_Stream_nat : Stream nat -> Stream nat -> Prop :=
| Bisimilar :
  forall (n1 n2 : nat) (ns1 ns2 : Stream nat),
    n1 = n2 ->
    bisimilar_Stream_nat ns1 ns2 ->
    bisimilar_Stream_nat (Cons n1 ns1) (Cons n2 ns2).
```

For example, let us prove that two ways of constructing the stream of even numbers are equivalent. In the first construction, a function that doubles its argument is mapped over the stream of natural numbers. In the second, a stream, starting from 0, is constructed by repeatedly applying a function that adds 2 to its argument. The following theorem states that the two resulting streams are bisimilar:

```
Theorem even_so :
  bisimilar_Stream_nat
    (map_Stream_nat (fun x => 2 * x) (make_Stream_nat S 0))
    (make_Stream_nat (fun x => 2 + x) 0).
```

To prove this theorem, we need to exhibit the head constructors of the two streams. If we can show that the first elements of the two streams are equal, then our proof reduces to showing that if the rest of the two streams are bisimilar, then the two streams are also bisimilar. The punchline of the proof will be to apply `Bisimilar`. As often, we first generalize the statement of this theorem into a master lemma, which we prove by coinduction:

```
1 subgoal

coIH : forall i : nat,
       bisimilar_Stream_nat
         (map_Stream_nat (fun x : nat => 2 * x) (make_Stream_nat S i))
         (make_Stream_nat (fun x : nat => 2 + x) (2 * i))
i : nat
============================
bisimilar_Stream_nat
  (Cons (2 * i)
     (map_Stream_nat (fun x : nat => 2 * x) (make_Stream_nat S (S i))))
  (Cons (2 * i)
     (make_Stream_nat (fun x : nat => 2 + x) (2 * S i)))
```

Fig. 1. Snapshot before applying `Bisimilar` in the proof of Lemma `even_so_aux`.

```
Lemma even_so_aux :
  forall i : nat,
   bisimilar_Stream_nat
     (map_Stream_nat (fun x => 2 * x) (make_Stream_nat S i))
     (make_Stream_nat (fun x => 2 + x) (2 * i)).
```

The course of this proof requires an auxiliary arithmetic lemma about distributing a multiplication over an addition:

```
Lemma even_so_aux_arithmetic :
  forall x : nat,
   2 * (S x) = 2 + 2 * x.
```

The proof of the master lemma is standard: we use the unfolding lemmas associated to `make_Stream_nat` and `map_Stream_nat` as rewriting rules to exhibit the head constructors of the two streams, and we use the auxiliary lemma as a rewriting rule to distribute the multiplication over the addition. We are then in position to apply `Bisimilar` (see Fig. 1), which creates two subgoals. The first subgoal pertains to the equality of the first elements of the two streams, and is solved by reflexivity, and the second subgoal pertains to the rest of the two streams, and is solved by applying the coinduction hypothesis:

```
Proof. (* of Lemma even_so_aux *)
  cofix coIH.
  intro i.
  rewrite -> (unfold_make_Stream_nat _ i).
  rewrite -> unfold_map_Stream_nat.
  rewrite -> (unfold_make_Stream_nat _ (2 * i)).
  rewrite <- even_so_aux_arithmetic.
  apply Bisimilar; [reflexivity | apply (coIH (S i))].
Qed.
```

Since their first operands are the same and their second are bisimilar, the two streams are bisimilar.

Prior to stating the master lemma, we first attempted to prove the theorem by exhibiting the successive head constructors of each stream. We then identified their pattern, and generalized it into the statement of the master lemma.

2.2. Stream processing and reasoning about it: partial sums

We map a stream to the stream of its partial sums by recursively traversing it with an accumulator:

```
CoFixpoint partial_sums_aux (a : nat) (ns : Stream nat) : Stream nat :=
  match ns with
    | Cons n ns' => Cons (n + a) (partial_sums_aux (n + a) ns')
  end.
```

As in Section 2.1, we equip `partial_sums_aux` with an unfolding lemma:

```
Lemma unfold_partial_sums_aux :
  forall (a n : nat) (ns' : Stream nat),
   partial_sums_aux a (Cons n ns') =
   Cons (n + a) (partial_sums_aux (n + a) ns').
```

This accumulator is initialized with `0`:

```
Definition partial_sums (ns : Stream nat) : Stream nat :=
  partial_sums_aux 0 ns.
```

To prove a theorem about `partial_sums`, we typically generalize it into a master lemma about `partial_sums_aux`, which we then apply to the initial value of the accumulator to prove the theorem, as in Section 2.1.

```
1 subgoal

coIH : forall i : nat,
       bisimilar_Stream_nat
          (partial_sums_aux (i ^ 2)
             (make_Stream_nat (fun x : nat => S (S x)) (1 + 2 * i)))
          (map_Stream_nat (fun x : nat => x ^ 2)
             (make_Stream_nat S (S i)))
i : nat
============================
bisimilar_Stream_nat
   (Cons (S i ^ 2)
       (partial_sums_aux (S i ^ 2)
          (make_Stream_nat (fun x : nat => S (S x)) (1 + 2 * S i))))
   (Cons (S i ^ 2)
       (map_Stream_nat (fun x : nat => x ^ 2)
          (make_Stream_nat S (S (S i)))))
```

Fig. 2. Snapshot before applying `Bisimilar` in the proof of Lemma `odd_adder_aux`.

2.3. *The odd little adder's discovery, formalized*

Let us formalize the odd little adder's discovery: partially summing the stream of odd numbers yields the stream of positive squares. In Coq:

```
Theorem odd_adder :
  bisimilar_Stream_nat
    (partial_sums (make_Stream_nat (fun x => 2 + x) 1))
    (map_Stream_nat (fun x => x ^ 2) (make_Stream_nat S 1)).
```

As before, we need to generalize the statement of this theorem into a master lemma that quantifies over `partial_sums_aux`'s accumulator. To this end, in the privacy of our Emacs buffer, we attempted to prove the theorem by exhibiting the successive head constructors of each stream. We then identified their pattern and—Eureka—generalized it into the following master lemma.

```
Lemma odd_adder_aux :
  forall i : nat,
   bisimilar_Stream_nat
     (partial_sums_aux (i ^ 2)
       (make_Stream_nat (fun x => 2 + x) (1 + 2 * i)))
     (map_Stream_nat (fun x => x ^ 2) (make_Stream_nat S (S i))).
```

As in Section 2.1, we proceed by coinduction towards applying `Bisimilar`. The course of the proof requires two auxiliary arithmetic lemmas about binomial expansion and about distributing a multiplication over an addition and reassociating the resulting addition:

```
Lemma binomial_expansion_2 :
  forall x : nat,
   S x ^ 2 = 1 + 2 * x + x ^ 2.


Lemma odd_adder_aux_arithmetic :
  forall x : nat,
   2 + (1 + 2 * x) = 1 + 2 * (S x).
```

Again, we use the unfolding lemmas associated to `make_Stream_nat` and `map_Stream_nat` as rewriting rules to exhibit the head constructors of the two streams, and we use the auxiliary lemmas as rewriting rules. We are then in position to apply `Bisimilar` (see Fig. 2), which creates two subgoals. The first subgoal pertains to the equality of the first elements of the two streams, and is solved by reflexivity, and the second subgoal pertains to the rest of the two streams, and is solved by applying the coinduction hypothesis:

```
Proof. (* of Lemma odd_adder_aux *)
  cofix coIH.
  intro i.
  rewrite -> (unfold_make_Stream_nat _ (1 + 2 * i)).
  rewrite -> unfold_partial_sums_aux.
  rewrite <- binomial_expansion_2.
  rewrite -> odd_adder_aux_arithmetic.
  rewrite -> (unfold_make_Stream_nat _ (S i)).
  rewrite -> unfold_map_Stream_nat.
  apply Bisimilar; [reflexivity | apply (coIH (S i))].
Qed.
```

The proof of the theorem follows. We unfold the definition of `partial_sums` to reveal `partial_sums_aux` and the initial value of its accumulator, and we apply the master lemma:

```
Proof. (* of Theorem odd_adder *)
  unfold partial_sums.
  apply (odd_adder_aux 0).
Qed.
```

2.4. *Stream processing and reasoning about it: dropping numbers*

We periodically drop a number in a stream using an auxiliary function `drop_aux` with a cyclic counter. In the common case where the counter is in the middle of its cycle, the head of the input stream is mapped to the head of the output stream and the counter is ticked. In the remaining case where the counter is at the end of its cycle, the head of the input stream is skipped and the counter is reset. The subsequent head in the input stream is also mapped to the head of the output stream, so that the recursive call to the auxiliary function is guarded by `Cons`, *Coq oblige*:

```
CoFixpoint drop_aux (period aux : nat) (ns : Stream nat) : Stream nat :=
  match ns with
    | Cons n ns' =>
      match aux with
        | 0 => match ns' with
                 | Cons n' ns'' =>
                   Cons n' (drop_aux period period ns'')
               end
        | S aux' => Cons n (drop_aux period aux' ns')
      end
  end.
```

We equip `drop_aux` with two unfolding lemmas, each of which is proved as the other unfolding lemmas in Section 2.1:

```
Lemma unfold_drop_aux_0 :
  forall (period' n n' : nat) (ns'' : Stream nat),
    drop_aux period' 0 (Cons n (Cons n' ns'')) =
    Cons n' (drop_aux period' period' ns'').

Lemma unfold_drop_aux_S :
  forall (period aux' n : nat) (ns' : Stream nat),
    drop_aux period (S aux') (Cons n ns') =
    Cons n (drop_aux period aux' ns').
```

For the period `0`, we vacuously define `drop` as the identity function over the input stream. Otherwise, for the period `1` (the second natural number), we filter out every second element, for the period `2` (the third natural number), we filter out every third element, etc. To this end, `drop` calls `drop_aux` with an initial counter corresponding to the period (`0` for the period `1`, `1` for the period `2`, etc.):

```
Definition drop (period : nat) (ns : Stream nat) : Stream nat :=
  match period with
    | 0 => ns (* vacuously *)
    | S period' => match ns with
                     | Cons n ns' => Cons n (drop_aux period' period' ns')
                   end
  end.
```

2.5. *Stream processing and reasoning about it: picking numbers*

To define Moessner's sieve, we also need a function `pick` complementing `drop`. So for example, applying `pick` to `0` (the first natural number) yields the same stream, applying `pick` to `1` (the second natural number) yields a stream containing every second element of the given stream, applying `pick` to `2` (the third natural number) yields a stream containing every third element of the given stream, etc. Like `drop`, `pick` uses an auxiliary definition `pick_aux`. For brevity, we do not display its definition and we do not state any property about it here, but both its definition, a property, and a proof of this property are included in the companion Coq files.

3. Moessner's sieve

We first state our thesis in Section 3.1. Then, for brevity, we define Moessner's sieve using primitive recursion in Section 3.2. We then prove our thesis in Section 3.3 and automate this proof in Section 3.4.

3.1. Our thesis

It is our thesis that the numbers dropped at each iteration of Moessner's sieve at any given rank are enumerated by a monomial at that rank indexed by the number of iterations:

```
Definition make_monomial (x rank index : nat) : nat :=
  (make_binomial_coefficient rank index) * x ^ index.
```

So for example, here is our thesis at rank 3 and at row 1:

```
Lemma moessner_aux_rank_3_row_1_in_long_hand :
  forall x : nat,
   bisimilar_Stream_nat
     (pick 2
       (partial_sums_aux (make_monomial x 3 1)
         (drop 3
           (partial_sums_aux (make_monomial x 3 0)
             (drop 4
               zeroes)))))
     (map_Stream_nat (fun y => make_monomial y 3 1)
       (make_Stream_nat S (S x))).
```

This lemma says that starting from the stream of zeroes, if we periodically drop its successive 5th elements, and then partially sum them over the first monomial at rank 3, we obtain a stream such that if periodically drop each of its successive 4th elements, and then partially sum them over the second monomial at rank 3, we obtain a stream whose every successive 3rd elements are enumerated by the stream of numbers enumerated by the second monomial at rank 3. More concisely, the numbers dropped at the third iteration of Moessner's sieve at rank 3 are enumerated by the third monomial at rank 3.

Here our thesis at rank 4 and at row 4:

```
Lemma moessner_aux_rank_4_row_4_in_long_hand :
  forall x : nat,
   bisimilar_Stream_nat
     (pick 0
       (partial_sums_aux (make_monomial x 4 4)
         (drop 1
           (partial_sums_aux (make_monomial x 4 3)
             (drop 2
               (partial_sums_aux (make_monomial x 4 2)
                 (drop 3
                   (partial_sums_aux (make_monomial x 4 1)
                     (drop 4
                       (partial_sums_aux (make_monomial x 4 0)
                         (drop 5
                           zeroes)))))))))))
     (map_Stream_nat (fun y => make_monomial y 4 4)
       (make_Stream_nat S (S x))).
```

Applying this lemma to 0 gives Moessner's theorem at rank 4.

3.2. A primitive–recursive definition of Moessner's sieve

Rather than writing in long hand each instance of Moessner's sieve at a given rank, let us define it once and for all by primitive recursion:

```
Definition Moessner_s_sieve_aux (rank index x : nat) : Stream nat :=
  pick (rank - index)
    (nat_rec (fun (_ : nat) => nat -> nat -> Stream nat)
              (fun period x =>
                zeroes)
              (fun j h period x =>
                partial_sums_aux (make_monomial x rank j)
                  (drop period
                  (h (S period) x)))
               (S index)
               (S (rank - index))
               x).


Definition Moessner_s_sieve (rank : nat) : Stream nat :=
 Moessner_s_sieve_aux rank rank 0.
```

We can verify that the primitive–recursive definition macro-expands to what we would write in long hand:

```
Example moessner_aux_rank_2_row_2 :
 forall x : nat,
  Moessner_s_sieve_aux 2 2 x =
  (pick 0
   (partial_sums_aux (make_monomial x 2 2)
    (drop 1
     (partial_sums_aux (make_monomial x 2 1)
      (drop 2
       (partial_sums_aux (make_monomial x 2 0)
        (drop 3
         zeroes))))))).
Proof.
 intro x.
 unfold Moessner_s_sieve_aux, nat_rec, nat_rect, minus.
 reflexivity.
Qed.
```

Compared to Section 3.1, we can now more concisely state our thesis at rank 3 and at row 1 as well as at rank 4 and at row 4, in constant space:

```
Lemma moessner_aux_rank_3_row_1 :
 forall x : nat,
  bisimilar_Stream_nat
   (Moessner_s_sieve_aux 3 1 x)
   (map_Stream_nat (fun y => make_monomial y 3 1)
    (make_Stream_nat S (S x))).


Lemma moessner_aux_rank_4_row_4 :
 forall x : nat,
  bisimilar_Stream_nat
   (Moessner_s_sieve_aux 4 4 x)
   (map_Stream_nat (fun y => make_monomial y 4 4)
    (make_Stream_nat S (S x))).
```

In fact, we can abstract away these lemmas into a generic function. Applying this function to a rank and a row, if the row is less or equal to the rank, gives the statement of each of these lemmas.

3.3. *Proving our thesis*

For any given rank, and for each index between 0 and that rank, the proof proceeds uniformly by unfolding the stream of zeroes and then successively unfolding each instance of `drop`, `drop_aux`, and `partial_sums` until the call to the `pick` function, which is also unfolded, giving the head constructor of the stream. Unfolding `make_Stream_nat` and `map_Stream_nat` gives the head constructor of the other stream. The heads of the two streams (two monomials) are Leibniz-equal. As for their tail, we need a series of arithmetic lemmas, as in Section 2.3, to put the first stream in shape for applying the coinduction hypothesis. The punchline of the proof is to apply `Bisimilar`, which creates two subgoals. The first subgoal pertains to the Leibniz equality of the first elements of the two streams, and is solved by reflexivity, and the second subgoal pertains to the rest of the two streams, and is solved by applying the coinduction hypothesis.

3.4. *Automating the proof of our thesis*

For any given rank, and for each index between 0 and that rank, the structure of the proof is so regular that we have written an iterative Ltac script parameterized by a rank and an index to perform the proof at that rank and at that index. The proof of each lemma is therefore a one-liner.

4. A left inverse of Moessner's sieve

In this section, we exploit our characterization of elided numbers with monomials to define a left inverse of Moessner's sieve. To this end, we first define a left inverse of `partial_sums` (Section 4.1) and then a left inverse of `drop` (Section 4.2). We then state the left inverse of Moessner's sieve at any given rank (Section 4.3).

4.1. A left inverse of partial sums

To invert `partial_sums`, we traverse its result, decrementing each of its elements by the previous one:

```
CoFixpoint partial_sums_op_aux (previous : nat) (ns : Stream nat) : Stream nat :=
  match ns with
    | Cons n ns' => Cons (n - previous) (partial_sums_op_aux n ns')
  end.

Definition partial_sums_op (ns : Stream nat) :=
  partial_sums_op_aux 0 ns.
```

Left inverseness is captured by the following theorem:

```
Theorem partial_sums_op_is_a_left_inverse_of_partial_sums :
  forall (ms ns : Stream nat),
   bisimilar_Stream_nat (partial_sums ms) ns ->
   bisimilar_Stream_nat ms (partial_sums_op ns).
```

To prove this theorem, we use a master lemma that quantifies the accumulator of `partial_sums_aux`.

4.2. A left inverse of dropping out

To invert `drop`, and given another stream, we mimic its definition and, instead of periodically skipping an element, we splice in the corresponding element from the other stream:

```
CoFixpoint drop_op_aux (count aux : nat) (ms ns : Stream nat) : Stream nat :=
  match aux with
    | 0 => match ms with
             | Cons m ms' =>
               Cons m (drop_op_aux count count ms' ns)
           end
    | S aux' => match ns with
                  | Cons n ns' =>
                    Cons n (drop_op_aux count aux' ms ns')
                end
  end.

Definition drop_op (period : nat) (ms ns : Stream nat) : Stream nat :=
  drop_op_aux period period ms ns.
```

Left inverseness is captured by the following theorem:

```
Theorem drop_op_is_a_left_inverse_of_drop :
  forall (period : nat) (ns ns_elided ns_kept : Stream nat),
   bisimilar_Stream_nat (pick (S period) ns) ns_elided ->
   bisimilar_Stream_nat (drop (S period) ns) ns_kept ->
   bisimilar_Stream_nat (drop_op (S period) ns_elided ns_kept) ns.
```

To prove this theorem, we use a master lemma that quantifies the accumulators of `drop_aux` and `drop_op_aux`.

4.3. A left inverse of Moessner's sieve at any given rank

Consider Moessner's sieve at rank 3. Starting from the stream of positive cubes, i.e., of the first monomial of $(x+1)^3$, if we

1. strip out the accumulated sums and splice in the successive instances of the second monomial of $(x+1)^3$, i.e., $3x^2$,
2. strip out the accumulated sums and splice in the successive instances of the third monomial of $(x+1)^3$, i.e., $3x$, and
3. strip out the accumulated sums and splice in the successive instances of the last monomial of $(x+1)^3$, i.e., 1,

we obtain the constant stream of ones:

```
Theorem moessner_op_3 :
 bisimilar_Stream_nat
   (drop_op 3
     (make_Stream_nat (fun x => make_monomial x 3 0) 1)
     (partial_sums_op
       (drop_op 2
         (make_Stream_nat (fun x => make_monomial x 3 1) 1)
         (partial_sums_op
           (drop_op 1
             (make_Stream_nat (fun x => make_monomial x 3 2) 1)
             (partial_sums_op
                (make_Stream_nat (fun x => make_monomial x 3 3) 1)))))))
   (make_Stream_nat (fun _ => 1) 0).
```

We have formalized the left inverse of Moessner's sieve and its assorted theorems as in Section 3, mutatis mutandis. This formalization works for any given rank.

5. Towards Long's theorem

Rather than starting from the stream `Cons 1 zeroes`, we can start from stream `Cons c zeroes`, for some `c`. Long's theorem [6] states that Moessner's sieve then yields $c1^n$, $c2^n$, $c3^n$, etc., i.e., the succession of positive natural numbers exponentiated by n and then multiplied by c. We have observed that the numbers that are successively elided are enumerated by the successive monomials of the binomial expansion of $(1+x)^n$, also multiplied by c. Given a rank and an index, the same proof scripts apply.

6. Conclusion

The odd little adder spoke carefully: "And so the property holds in general because it does for each natural number (and vice versa) and because natural numbers are constructed in one and only one way. Right?" He wanted to make sure. "Right," he was answered. So he continued: "I can see that the sieve can give not only squares, but also cubes, etc." And, after a pause, he added[2]: "But *why* does it give squares, cubes, etc.?" "Hum," he was answered. "Perhaps we should tell you about Newton's binomial expansion and Pascal's triangle."

In this article, we have characterized the elided numbers in Moessner's sieve at rank n as enumerating the successive monomials in the binomial expansion of $(1+x)^n$. The last numbers are therefore the successive exponents of x—i.e., 1^n, 2^n, 3^n, etc.—which is Moessner's theorem at that rank.[3] We have then exploited this characterization to define a left inverse of Moessner's sieve.

What we have not done here, however, is to show formally that the characterization holds not just for any given rank, but for any rank at all. Since Moessner's theorem at any rank does not depend on Moessner's theorem at any other rank, we are now working on formalizations of Moessner's theorem and of Long's theorem that are universally quantified by a rank.

Acknowledgements

This work was partly carried out during two visits of the third author to the second author, in the spring and in the summer of 2012. We are grateful to Irène Danvy, Mayer Goldberg, Kent Grigo, Julia Lawall, Filip Sieczkowski, Peter Urbak, and Ian Zerny for comments on a preliminary version, as well as to the anonymous reviewers. Special thanks to the editors for their encouragement and for their time management.

And of course, many happy returns to Glynn!

References

[1] Yves Bertot, Pierre Castéran, Interactive Theorem Proving and Program Development, Springer, 2004.
[2] Mark Bickford, Dexter Kozen, Alexandra Silva, Formalizing Moessner's theorem in Nuprl, http://www.nuprl.org/documents/Moessner/, August 2011.
[3] Ralf Hinze, Scans and convolutions – a calculational proof of Moessner's theorem, in: Sven-Bodo Scholz, Olaf Chitil (Eds.), Implementation and Application of Functional Languages, 20th International Workshop, IFL 2008, Hatfield, UK, September 2008, in: Lecture Notes in Computer Science, vol. 5836, Springer, 2008, pp. 1–24.
[4] Ralf Hinze, Concrete stream calculus – an extended study, J. Funct. Programming 20 (5–6) (2011) 463–535.

[2] What else? He is a little adder.
[3] "*That's* why."—Jasmine Dubrow

[5] Dexter Kozen, Alexandra Silva, On Moessner's theorem, Technical report, Computing and Information Science, Cornell University, June 2011.
[6] Calvin T. Long, On the Moessner theorem on integral powers, Amer. Math. Monthly 73 (8) (1966) 846–851.
[7] Calvin T. Long, Mathematical excitement—the most effective motivation, Math. Teach. 75 (5) (1982) 413–415.
[8] Calvin T. Long, Strike it out—add it up, Math. Gaz. 66 (428) (1982) 273–277.
[9] Calvin T. Long, A note on Moessner's process, Fibonacci Quart. 24 (4) (1986) 349–355.
[10] Zohar Manna, Richard J. Waldinger, Toward automatic program synthesis, Commun. ACM 14 (3) (1971) 151–165.
[11] Alfred Moessner, Eine Bemerkung über die Potenzen der natürlichen Zahlen, Sitzungsber. - Bayer. Akad. Wiss., Math.-Nat. Kl. 29 (3) (March 1951) 9.
[12] Milad Niqui, Jan Rutten, A proof of Moessner's theorem by coinduction, High.-Order Symbol. Comput. 24 (3) (2011) 191–206.
[13] Ivan Paasche, Ein neuer Beweis des Moessnerschen Satzes, Sitzungsber. - Bayer. Akad. Wiss., Math.-Nat. Kl. 30 (1) (February 1952) 1–5.
[14] Oskar Perron, Beweis des Moessnerschen Satzes, Sitzungsber. - Bayer. Akad. Wiss., Math.-Nat. Kl. 29 (4) (May 1951) 31–34.
[15] Hans Salié, Bemerkung zu einem Satz von, Sitzungsber. - Bayer. Akad. Wiss., Math.-Nat. Kl. 30 (2) (February 1952) 7–11.
[16] Saed Samadi, M. Omair Ahmad, M.N.S. Swamy, Multiplier-free structures for exact generation of natural powers of integers, in: International Symposium on Circuits and Systems (ISCAS 2005), Kobe, Japan, IEEE, May 2005, pp. 1146–1149.
[17] Davide Sangiorgi, Introduction to Bisimulation and Coinduction, Cambridge University Press, Cambridge, England, 2012.
[18] John G. Slater, Strike it out—some exercises, Math. Gaz. 67 (442) (1983) 288–290.
[19] Glynn Winskel, The Formal Semantics of Programming Languages, Foundation of Computing Series, MIT Press, 1993.
[20] Jan van Yzeren, A note on an additive property of natural numbers, Amer. Math. Monthly 66 (1) (1959) 53–54.