Moessner's Theorem: an exercise in coinductive reasoning in Coq

Robbert Krebbers[1], Louis Parlant[2], and Alexandra Silva[3]*

[1] Aarhus University, Denmark
[2] École Normale Supérieure de Lyon, France
[3] University College London, United Kingdom

Dedicated to Frank de Boer on the occasion of his 60^{th} birthday.

Abstract. Moessner's Theorem describes a construction of the sequence of powers $(1^n, 2^n, 3^n, \dots)$, by repeatedly dropping and summing elements from the sequence of positive natural numbers. The theorem was presented by Moessner in 1951 without a proof and later proved and generalized in several directions. More recently, a coinductive proof of the original theorem was given by Niqui and Rutten. We present a formalization of their proof in the Coq proof assistant. This formalization serves as a non-trivial illustration of the use of coinduction in Coq. During the formalization, we discovered that Long and Salié's generalizations could also be proved using (almost) the same bisimulation.

1 Introduction

Coinduction has grown in the last years as the prime principle to prove properties about dynamical and concurrent systems or, in general, structures that exhibit circularity. Formalizations of coinduction are becoming common in most proof assistants but the use thereof is not yet widespread, often due to the lack of good examples balancing expressivity and simplicity to be suitable tutorials for new users. This paper sets itself to provide such an example tutorial of formalized coinduction. Formal methods, concurrency, and verification have been central topics in Frank's research and in the last decade he was exposed (though not intentionally!) to coinduction frequently. We dedicate to Frank this paper on formalizing a result about Frank's favorite coinductive object – streams.

Streams constitute the most basic example of infinite objects and are often used to illustrate the use of coinduction to prove equivalence of algorithms producing infinite objects. A more elaborate example of the use of coinduction to prove the correctness of an algorithm that produces infinite objects is provided by Niqui and Rutten's proof of Moessner's Theorem [13].

Moessner's Theorem describes a procedure for constructing the stream of successive exponents $(1^n, 2^n, 3^n, \dots)$, for every $n \geq 1$, with several steps of dropping

* The work in this paper was developed when all authors were at the Radboud University, The Netherlands.

and summing elements of the stream of positive natural numbers. This procedure is quite simple: let us show the result for $n = 3$. Starting with the sequence of positive naturals $(1, 2, 3, 4, 5, 6, 7, 8, \dots)$, one drops every *third* element to obtain the stream $(1, 2, 4, 5, 7, 8, \dots)$. Then one computes the stream of the partial sums by adding to every element all the previous ones:

$$(1, 1+2, 1+2+4, 1+2+4+5, 1+2+4+5+7, \dots) = (1, 3, 7, 12, 19, \dots)$$

Then, one drops every *second* element of the latter sequence, giving rise to $(1, 7, 19, \dots)$, and finally by taking partial sums, one gets: $(1, 8, 27, \dots)$. The resulting stream contains indeed the expected elements: $(1^3, 2^3, 3^3, \dots)$.

This result holds for any n: drop every n-th element of the sequence of positive naturals, then form partial sums, and then start again dropping every $(n-1)$-th element and summing, and proceed recursively. This process creates the stream of all positive naturals to the power of n: $(1^n, 2^n, 3^n, 4^n, 5^n, \dots)$.

The above algorithm/procedure can easily be described as a functional program that takes n as a parameter. Moessner's Theorem now corresponds to the question of whether this program yields the stream $(1^n, 2^n, 3^n, 4^n, 5^n \dots)$ for each $n \geq 1$. Since the stream $(1^n, 2^n, 3^n, 4^n, 5^n \dots)$ is a functional program in itself, Moessner's Theorem can be proven by showing equivalence of these programs. Because these programs produce streams, the obvious technique to prove equivalence is to use *coinduction*. This was observed by Niqui and Rutten who provided a bisimulation witnessing the equivalence of these programs [13].

Related work. Moessner's construction has attracted much attention over the years. The theorem was only conjectured by its discoverer [12]. The first proof was given shortly thereafter by Perron [17] (who, curiously, was the editor of the journal where the conjecture was submitted). The theorem was then the subject of several popular accounts and generalizations [4,8,11,20,14,15,16].

Paasche [14,15,16] generalized it by allowing the dropping intervals to increase at each step. This led to the construction of the stream containing the factorials and super-factorials. Long [10,11] and Salié [20] also generalized Moessner's result to apply to the situation in which the initial sequence is not the sequence of successive integers $(1, 2, 3, \dots)$ but the arithmetic progression $(a, d + a, 2d + a, \dots)$. They showed that the final sequence obtained by the Moessner construction is $(a \cdot 1^{n-1}, (d + a) \cdot 2^{n-1}, (2d + a) \cdot 3^{n-1}, \dots)$.

Very recently, Hinze [7] and Niqui and Rutten [13] have given proofs involving concepts from functional programming, respectively calculational scans and the coalgebra of streams. The proof of Hinze covers Moessner's and Paasche's results whereas Niqui and Rutten's proof only covers the original Moessner's Theorem.

Kozen and the third author [9] have provided an algebraic proof that has the advantage of covering all the results mentioned above and opened the door to new generalizations of Moessner's original result. The foundations of this proof were formalized in NUPRL by Bickford *et al.* [2].

Clausen *et al.* [3] have also provided a formalization of Moessner's theorem in COQ, but their approach is very different from ours. Our result is more general

and applies to $(1^n, 2^n, 3^n, \dots)$ for any $n \geq 1$, whereas they provide a Coq tactic that *generates* a theorem for any given n by macro expansion. We furthermore also provide a proof of Long and Salié's generalization, that is both more general, and follows as a mere consequence of the original Moessner's Theorem.

Urbak [23] extended the results of Clausen *et al.* in his MSc thesis by exploring Moessner's theorem in a very general setting. Long and Salié's generalization is also a consequence of his work.

Contrary to the aforementioned Coq formalizations, we have setup our Coq development in such a way that it matches common mathematical practice in coinduction, as for example being used by Rutten [18]. We have abstracted from Coq's implementation of coinduction as much as possible by providing an abstraction on top of it to avoid for example guardedness issues in proofs. Also, we have made heavy use of Coq's notations machinery to obtain notations close to those on paper, and have automated parts of the proof that one would omit on paper too. As a result, we were able to formalize the proof of Niqui and Rutten in a very compact and concise way that is close to its original presentation. Our Coq development is 20 times shorter, in terms of lines of code, than Urbak's.

Contribution. We set ourselves to the quest of formalizing Niqui and Rutten's proof in the Coq proof assistant [5]. The interest in doing so is four-fold.

- On the one hand, as with every formalization, one is forced to go through all details of the *pen-and-paper* proof and potentially uncover flaws or omissions.
- On the other hand, and of more interest to us, coinduction in Coq is not widely used and good (tutorial) examples are lacking. Bisimulation proofs are very mechanical and particularly suited for automation/formalization in proof assistants. Hence, we hope that the present example can serve as non-trivial teaching/illustration material of a proof by coinduction in Coq.
- There is often just a shallow correspondence between formalizations and their original mathematical texts. We show that this is not necessarily the case by defining suitable abstractions. In particular, we abstract from Coq's internals for coinduction as much as possible. As a result, our formalization corresponds well to the paper by Niqui and Rutten, and is very compact.
- Lastly, in the process of formalizing Niqui and Rutten's proof, we uncovered a simple proof of Long [10,11] and Salié's [20] generalization. Though (once done) the generalization is not at all complicated, it was surprising to us that the extended version is just a corollary of the original Moessner's Theorem, and that the the bisimulation did not have to be modified. The Coq formalization was achieved with a simple extra lemma.

Our Coq code is available at `https://github.com/robbertkrebbers/moessner`.

2 Streams and Coinduction

In the construction of Moessner's Theorem, streams and operations on them (in particular, drop and sum) play a central role. The set of *streams* A^ω with elements in A can be formally defined as $A^\omega = \{s \mid s \colon \mathbb{N} \to A\}$.

We denote the n-th element of the stream s by $s(n)$. Given a stream $s = (a_0, a_1, a_2, a_3, \dots)$, we call $s(0) = a_0$ the *head* of the stream, and $(a_1, a_2, a_3, \dots)$ the *tail* of the stream, which we denote by s'. The operations of head and tail define the following structure on the set of streams:

$$c\colon A^\omega \longrightarrow A \times A^\omega \qquad c(s) = (s(0), s'). \tag{1}$$

The functor F corresponding to the above structure is $F(X) = A \times X$. The set of streams A^ω is the greatest fixpoint of this functor. That is in essence why streams are *coinductive* type, in contrast with lists, which are the least fixpoint of the functor $G(X) = 1 + A \times X$.

In Coq we define streams using the latter view as a coinductive type instead of the functional view $\{s \mid s\colon \mathbb{N} \to A\}$. The coinductive view on streams allows for a simple and elegant definition of operations, as well as for proofs of properties on them. The coinductive approach to infinite datatypes enables a uniform extension to more complex types, such as infinite trees, λ-terms, automata, *etc.* [19].

Streams are the simplest examples of coalgebras and proofs of stream equality are prime illustrations of the power of the coinduction proof principle. Since in this paper we will only deal with streams, we will be introducing all general concepts concretely in this context. The proof of Moessner's Theorem is a beautiful example of *concrete coalgebra*.

Definition 1. *A relation $R \subseteq A^\omega \times A^\omega$ is a* bisimulation *if for every $(s,t) \in R$ it holds that $s(0) = t(0)$ and $(s', t') \in R$.*

The following theorem states the coinduction proof principle for streams, which enables one to prove equality of streams just by exhibiting a bisimulation relation containing the pair consisting of these two streams.

Theorem 1 (Coinduction Principle). *Let $R \subseteq A^\omega \times A^\omega$ be a bisimulation. For all $s, t \in A^\omega$ we have that $(s,t) \in R$ implies $s = t$.*

3 Basic operations and theorems on streams in Coq

In this section we describe the Coq definitions of operations on streams that are needed to formalize Moessner's Theorem. Also, we describe the basic theorems and Coq infrastructure that we use for the formalization. In order to get started, we first define the type `Stream A` of streams A^ω with elements of type A.

```
CoInductive Stream (A : Type) : Type :=
  SCons : A → Stream A → Stream A.
Arguments SCons {_} _ _. (* Setup implicit arguments so Coq infers the
type [A] of [SCons : ∀ A : Type, A → Stream A → Stream A]. *)
Infix ":::" := SCons.
```

This definition resembles the well-known inductive definition of lists, but instead of the keyword `Inductive` we use the keyword `CoInductive`. Furthermore, note that `:::` is the inverse map of the structure map on the set of streams given

by head and tail, *cf.* (1) on page 4, and the keyword `CoInductive` is taking the greatest fixpoint of the functor $F(X) = A \times X$, as described in Section 2.

The `CoFixpoint` command is used to create corecursive definitions:

```
CoFixpoint repeat {A} (x : A) : Stream A := x ::: #x
where "# x" := (repeat x).
```

The stream `#`x represents the *constant stream* $(x, x, x, \dots)$ that Niqui and Rutten denote by $\overline{x}$. Whereas recursive definitions in Coq should be *terminating*, corecursive definitions should be *productive*. Intuitively this means that given a term of coinductive type (and in particular a `CoFixpoint`), it will always produce a constructor. The following is rejected by Coq because this is not the case.

```
Fail CoFixpoint bad : Stream False := bad.
```

Since productivity is undecidable, corecursive definitions in Coq should satisfy a decidable syntactical criterion (so as to enable decidable type checking) that guarantees productivity. This criterion is called the *guard condition*. Over simplified, this means that a `CoFixpoint` should have the following shape:

```
CoFixpoint f p⃗ : Stream A := x0 ::: x1 :::  ... ::: xn ::: f q⃗.
```

with $0 < n$. The definition of `#x` satisfies this condition, but `bad` does not.

Although the guard condition ensures that terms of coinductive type always produce a constructor, Coq's computation rules do not allow `CoFixpoint` definitions to reduce. For example, `# 10` does not reduce to `10 ::: #x`. If it would, this process could be repeated infinitely many times, and would destroy the property that all computations in Coq terminate. Instead, computation of coinductive types is performed lazily, and a `CoFixpoint` definition is only allowed to reduce whenever it is the operand of a pattern match construct.

Pattern matching can be used to decompose coinductive types. For streams, this mechanism allows us to define the common destructors `head` and `tail`.

```
Definition head {A} (s : Stream A) : A := match s with x ::: _ ⇒ x end.
Definition tail {A} (s : Stream A) : Stream A :=
  match s with _ ::: s ⇒ s end.
Notation "s ‘" := (tail s).
```

We use the notation `s‘` for `tail s` so as to resemble the presentation of Niqui and Rutten. Of course, Coq allows us to write expressions like `s‘‘` to denote the second tail of `s`. Notice that the term `head (#10)` indeed reduces to `10` because the `CoFixpoint` definition now becomes the operand of a pattern match construct.

We will not be using explicit pattern matching on streams anymore, and define everything in terms of `head` and `tail`. For example, see below the functions `map` and `zip_with` which lift functions on individual elements to whole streams.

```
CoFixpoint map {A B} (f : A → B) (s : Stream A) : Stream B :=
  f (head s) ::: map f (s‘).
CoFixpoint zip_with {A B C} (f : A → B → C)
    (s : Stream A) (t : Stream B) : Stream C :=
  f (head s) (head t) ::: zip_with f (s‘) (t‘).
```

3.1 Stream equality, bisimulation, and coinduction

In order to support algebraic reasoning about streams, we need a notion that expresses that streams are element-wise equal. Since no finite expansion of the streams `#f x` and `map f (#x)` lead to equal terms, Coq's notion of Leibniz equality `=` is too strong to accurately capture stream equality [6,1]. Therefore, we use the following coinductively defined relation of *bisimilarity*[4]:

```
CoInductive equal {A} (s t : Stream A) : Prop :=
  make_equal : head s = head t → s‘ ≡ t‘ → s ≡ t
where "s ≡ t" := (@equal _ s t).
```

Since bisimilarity is defined as a coinductive type, proving that two streams are bisimilar corresponds to constructing a corecursive definition by the Curry-Howard correspondence (programs as proofs). For example, we can construct a proof of `#f x ≡ map f (#x)` by providing an explicit proof term as follows:

```
CoFixpoint repeat_map {A B} (f : A → B) x : #f x ≡ map f (#x) :=
  make_equal (#f x) (map f (#x)) eq_refl (repeat_map f x).
```

Here, `eq_refl` is a proof of `f x = f x`, and thus a proof of `head (#f x) = head (map f (#x))` by convertibility. Clearly, proving such properties by providing an explicit proof term is inconvenient, and should be avoided in practice.

Coq's native support for coinductive proofs is not as good as its support for inductive proofs. There is just the primitive `cofix` tactic, which does not protect one from creating proofs that do not satisfy the guard condition. If a proof does not satisfy the guard condition, the proof will only be rejected when one closes the proof using `Qed` (that is when the proof is being checked by the kernel). This is different from the `induction` tactic, which cannot be used wrongly. Let us give a demonstration of the `cofix` tactic.

```
Lemma repeat_map x : #f x ≡ map f (#x).
Proof.
  cofix CH.
  (* We get a hypothesis [CH : #f x ≡ map f (#x)]. We should use it in
     such a way that the generated proof term is guarded. *)
  apply make_equal.
  * (* Prove that the heads are equal: [head (#f x) = head (map f (#x))]
    This holds by computation, so [reflexivity] will succeed. *)
    reflexivity.
  * (* Prove that the tails are equal: [(#f x)‘ ≡ (map f (#x))‘] *)
    (* Unfold the definitions to obtain [#f x ≡ map f (#x)] *)
    (* NB: the exclamation mark ! performs a rewrite as many times as
       possible (but at least once) *)
    rewrite map_tail, !repeat_tail.
    (* Use the corecursive assumption [CH]. *)
    exact CH.
Qed.
```

[4] We use Leibniz equality for the heads because we only deal with streams of integers. In general, for example to consider streams of streams, this is still too restrictive.

In the above proof, it would be appealing to use the hypothesis `CH` straightaway. Of course, the generated proof term would not be guarded, and will therefore be rejected whenever we type `Qed`. Since we have to be extremely careful when to use the hypothesis generated by the `cofix` tactic, many tactics for automation cannot be used for coinductive proofs because they will use hypotheses eagerly and thus likely break the guard condition. Therefore we will look at two alternative approaches to proving stream equality.

The first approach is to define a *stream bisimulation* relation (see Definition 1), and then prove the coinduction proof principle (see Theorem 1). This is the core of the coinductive proof of Moessner's Theorem by Niqui and Rutten.

```
Definition bisimulation {A} (R : relation (Stream A)) : Prop :=
  ∀ s t, R s t → head s = head t ∧ R (s‘) (t‘).
Lemma bisimulation_equal {A} (R : relation (Stream A)) s t :
  bisimulation R → R s t → s ≡ t.
```

Instead of having to produce a proof-term that satisfies the guard condition, one has to define a suitable bisimulation relation, and the problem of guardedness has moved once and for all to the proof of `bisimulation_equal`.

Another approach is to view streams `Stream A` as functions `nat → A` (as we have initially introduced streams in Section 2). The function s `!!` i gives the ith element $s(i)$ of the stream s. It is straightforward to prove that streams are bisimilar if and only if they are element-wise equal using the `!!` function.

```
Fixpoint elt {A} (s : Stream A) (i : nat) : A :=
  match i with O ⇒ head s | S i ⇒ s‘ !! i end
where "s !! i" := (elt s i).
Lemma equal_elt {A} (s t : stream A) : s ≡ t ↔ ∀ i, s !! i = t !! i.
```

For many streams `!!` enjoys nice properties. The lemma `equal_elt` is thus often useful to prove stream equality. For example, using the lemmas:

```
Lemma repeat_elt {A} (x : A) i : #x !! i = x.
Lemma map_elt {A B} (f : A → B) s i : map f s !! i = f (s !! i).
```

we can give yet another proof of `#f x ≡ map f (#x)`.

```
Lemma repeat_map x : #f x ≡ map f (#x).
Proof.
  apply equal_elt. intros i. rewrite map_elt, !repeat_elt. reflexivity.
Qed.
```

By using `equal_elt`, we have to prove that `#f x !! i = map f (#x)!! i` for any `i`. This trivially follows from the lemmas above.

3.2 Setoids

In order to enable algebraic reasoning about streams, we should be able to rewrite using bisimilarity. We thus prove that `equal` is an equivalence relation.

```
Instance equal_equivalence {A} : Equivalence (@equal A).
```

We use the `Instance` keyword instead of the `Lemma` keyword to register this fact with Coq's setoid machinery [21]. The setoid machinery uses Coq's type classes [22] under water, but we will not detail that here.

Of course, rewriting with bisimilarity gives rise to side-conditions: rewriting a subterm is allowed only if the subterm is an argument of a function that has been proven to *respect* bisimilarity. For the case of `:::` this means that `s ≡ t` implies `x ::: s ≡ x ::: t`. In Coq this property can be expressed compactly by the following notation:

```
Instance SCons_proper {A} (x : A) : Proper (equal ⟹ equal) (SCons x).
```

This notation should be read as: if the arguments of `SCons x` are bisimilar, then so are the results. The arrow ⟹ should not be confused with the arrow → for function types. A property like the above must be proved for each function in whose arguments we wish to rewrite. For example:

```
Instance head_proper {A} : Proper (equal ⟹ eq) (@head A).
Instance tail_proper {A} : Proper (equal ⟹ equal) (@tail A).
Instance elt_proper {A} : Proper (equal ⟹ eq ⟹ eq) (@elt A).
```

3.3 Ring structure

We define the operations for element-wise addition, multiplication, and subtraction, by lifting the operations on integers using `zip_with` and `map`.

```
Infix "⊕" := (zip_with Z.add). (* addition *)
Infix "⊖" := (zip_with Z.sub). (* subtraction *)
Infix "⊙" := (zip_with Z.mul). (* multiplication *)
Notation "⊖ s":= (map Z.opp s). (* additive inverse *)
```

Together with the constant streams `#0` and `#1`, these operations introduce a ring structure on streams. To prove this result, we use the lemma `equal_elt` that relates bisimilarity to element-wise equality.

```
Lemma stream_ring_theory :
  ring_theory (#0) (#1) (zip_with Z.add) (zip_with Z.mul)
    (zip_with Z.sub) (map Z.opp) equal.
Add Ring stream : stream_ring_theory.
```

The command `Add Ring stream : stream_ring_theory` registers this fact, so that ring equations over streams can be solved automatically using the `ring` tactic. Automation for solving ring equations will be used heavily in Section 4.

```
Lemma Smult_plus_distr_r s t u : (t ⊕ u) ⊙ s ≡ (t ⊙ s) ⊕ (u ⊙ s).
Proof. ring. Qed.
```

The repeated multiplication defines the *stream power*, written $s^{\langle n \rangle}$:

```
Fixpoint Spow (s : Stream Z) (n : nat) : Stream Z :=
  match n with 0 ⇒ #1 | S n ⇒ s ⊙ s ^^ n end
where "s ^^ n" := (Spow s n).
```

3.4 Specific stream operations

In the last part of this section we define stream operations that are specifically used for Moessner's Theorem.

The stream of positive natural numbers `nats` is defined as the unique solution of the equations: $\texttt{nats}(0) = 1$ and $\texttt{nats}' = \overline{1} \oplus \texttt{nats}$. In order to define this stream in Coq, we define a more general notion that makes use of an accumulator. The definition `Sfrom` i represents the stream $(i, 1+i, 2+i, \dots)$.

```
CoFixpoint Sfrom (i : Z) : Stream Z := i ::: Sfrom (1 + i).
Notation nats := (Sfrom 1).
```

The equation of `nats` without an accumulator as given by Niqui and Rutten is not accepted by Coq because the co-recursive call to `nats` is hidden behind the $\oplus$ operation. This is not allowed by the guard condition.

```
Fail CoFixpoint nats : Stream Z := 1 ::: #1 ⊕ nats. (* Not allowed *)
```

Although this definition is rejected by Coq, we can still prove that the heads and tails of our definition satisfy the desired equations with respect to head and tail. This allows reasoning in the same way as on paper.

```
Lemma Sfrom_tail n : (Sfrom n)‘ ≡ #1 ⊕ Sfrom n.
```

Another operation that arises in the Moessner construction as described in the introduction is *partial sums* of a stream. This operation is informally defined by:

$$\Sigma\,(s_0, s_1, s_2, \dots) = (s_0, s_0 + s_1, s_0 + s_1 + s_2, \dots)$$

and formally by the equations $(\Sigma\, s)\,(0) = s(0)$ and $(\Sigma\, s)' = \overline{s} \oplus \Sigma\, s'$. In order to define the partial sums in Coq we again need to make use of an accumulator, and prove that the definition satisfies the desired equation.

```
CoFixpoint Ssum (i : Z) (s : Stream Z) : Stream Z :=
  head s + i ::: Ssum (head s + i) (s‘).
Notation "’Σ’ s" := (Ssum 0 s).
Lemma Ssum_tail s : (Σ s)‘ ≡ #head s ⊕ Σ s‘.
```

The last operation we need to define the Moessner construction is *dropping*. We define a family of drop operators $\mathrm{D}_k^i \colon A^\omega \to A^\omega$ as the solution of:

$$(\mathrm{D}_k^{i+1}\, s)(0) = s(0) \quad (\mathrm{D}_k^{i+1}\, s)' = \mathrm{D}_k^i\, s' \quad (\mathrm{D}_k^0\, s)(0) = s(1) \quad (\mathrm{D}_k^0\, s)' = \mathrm{D}_k^{k-2}\, s''.$$

The drop operator $\mathrm{D}_k^i\, s$ repeatedly drops the i-th element of every block of k elements of s. For example, $\mathrm{D}_3^1\, s = (s(0), s(2), s(3), s(5), s(6), s(8), \dots)$. We use the notation `D@{`i`,`k`}` s to denote this operation in Coq.

```
CoFixpoint Sdrop {A} (i k : nat) (s : Stream A) : Stream A :=
  match i with
  | O ⇒ head (s‘) ::: D@{k-2,k} s‘‘
  | S i ⇒ head s ::: D@{i,k} s‘
  end
where "D@{ i , k } s" := (Sdrop i k s).
```

This definition is identical to the definition of Niqui and Rutten, but whereas they require $2 \leq k$ and $0 \leq i < k$, we allow any k and i (subtraction of naturals $i - j$ is a total Coq function that yields 0 in case $i < j$).

4 A formalized proof of Moessner's Theorem

We are now ready to formulate Moessner's Theorem using the stream operations that we have defined. For the case $n = 3$, as presented in the introduction, the theorem boils down to the stream equation $\Sigma\, \mathrm{D}_2^1\, \Sigma\, \mathrm{D}_3^2\, \mathtt{nats} = \mathtt{nats}^{\langle 3 \rangle}$.

The general case is slightly more involved (mainly due to the amount of indices), but still mirrors very well the informal construction:

$$\Sigma\, \mathrm{D}_2^1\, \Sigma\, \mathrm{D}_3^2 \cdots \Sigma\, \mathrm{D}_n^{n-1}\, \mathtt{nats} = \mathtt{nats}^{\langle n \rangle}.$$

Niqui and Rutten start from the stream of ones, $\overline{1}$, and define an operator combining summing and dropping, namely $\Sigma_n^k = \Sigma\, \mathrm{D}_n^k$, which leads to a shorter formulation of the theorem: $\Sigma_2^1\, \Sigma_3^2 \cdots \Sigma_{n+1}^n\, \overline{1} = \mathtt{nats}^{\langle n \rangle}$. The simplification by Niqui and Rutten of not starting from the stream `nats` of positive natural numbers but from the stream $\overline{1}$ of ones is justified by the equation $\mathtt{nats} = \Sigma\, \overline{1}$.

In order to state Moessner's Theorem formally we introduce the Coq definition Σ`@{`i`,`k`,`n`}` s that recursively defines the sequence $\Sigma_k^i \cdots \Sigma_{n+k}^{n+i}\, s$.

```
Definition Ssigma (i k : nat) (s : Stream Z) : Stream Z := Σ D@{i,k} s.
Notation "Σ @{ i , k } s" := (Ssigma i k s).
Fixpoint Ssigmas (i k n : nat) (s : Stream Z) : Stream Z :=
  match n with
  | O ⇒ Σ @{i,k} s
  | S n ⇒ Σ @{i,k} Σ @{S i,S k,n} s
  end
where "Σ @{ i , k , n } s" := (Ssigmas i k n s).
```

Moessner's Theorem is then stated in Coq as:

```
Theorem Moessner n : Σ @{1,2,n} #1 ≡ nats ^^ S n.
```

4.1 The bisimulation relation

In order to prove Moessner's Theorem by coinduction, we define the bisimulation relation of Niqui and Rutten using an inductively defined relation.

```
Inductive Rn : relation (Stream Z) :=
  | Rn_sig1 n : Rn (Σ @{1,2,n} #1) (nats ^^ S n)
  | Rn_sig2 n : Rn (Σ @{0,2,n} #1) (nats ⊙ (#1 ⊕ nats) ^^ n)
  | Rn_refl s : Rn s s
  | Rn_plus s1 s2 t1 t2 : Rn s1 t1 → Rn s2 t2 → Rn (s1 ⊕ s2) (t1 ⊕ t2)
  | Rn_mult n s t : Rn s t → Rn (#n ⊙ s) (#n ⊙ t)
  | Rn_eq s1 s2 t1 t2 : s1 ≡ s2 → t1 ≡ t2 → Rn s1 t1 → Rn s2 t2.
```

The relation `Rn` is nearly a literate Coq translation of the bisimulation relation given by Niqui and Rutten. There are three small differences:

- Niqui and Rutten use indexes that count from 1 instead of 0. When working in a formal system, this is inconvenient, as it leads to many side-conditions.
- Since we consider streams of integers instead of streams of naturals (to make the generalizations in Section 5 possible), we had to explicitly close the bisimulation relation under scalar multiplication (using the constructor `Rn_mult`).
- Because we use bisimilarity to express stream equality, we had to close the bisimulation relation under it (using the constructor `Rn_eq`).

4.2 Proof outline

In what follows we show that `Rn` is a bisimulation relation, from which Moessner's Theorem is a direct consequence.

```
Lemma bisimulation_Rn : bisimulation Rn.
Theorem Moessner n : Σ@{1,2,n} #1 ≡ nats ^^ S n.
Proof.
  eapply bisimulation_equal, Rn_sig1.
  apply bisimulation_Rn.
Qed.
```

In order to prove the lemma `bisimulation_Rn`, we have to prove that `Rn s t` implies `head s = head t` and `Rn (s')(t')`. This is proven by induction on the derivation of `Rn`. There are two interesting cases:

1. The case corresponding to the constructor `Rn_sig1` for which we have to show that `Rn (Σ@{1,2,n} #1) (nats ^^ S n)` implies:

   `head (Σ@{1,2,n} #1) = (nats ^^ S n)`
   and
   `Rn ((Σ@{1,2,n} #1)') ((nats ^^ S n)')`.

   This case is covered by [13, Proposition 5.1-5.2] and formalized in Section 4.3.
2. The case corresponding to `Rn_sig2` involving `Rn (Σ@{0,2,n} #1) (nats ⊙ (#1 ⊕ nats) ^^ n)`. This case is covered by [13, Proposition 5.3-5.4] and formalized in Section 4.4.

The other cases follow from simple equational reasoning.

4.3 Case `Rn (Σ@{1,2,n} #1) (nats ^^ S n)`

In order to formalize the first case, we need to relate the heads and tails of the streams `Σ@{1,2,n} #1` and `nats ^^ S n`. This case involves proving the equations below [13, Proposition 5.1-5.2]:

$$\texttt{head (Σ@\{1,2,n\} \#1)} = 1 = \texttt{head (nats ^^ S n)}$$

$$\texttt{(Σ@\{1,2,n\} \#1)'} \equiv \texttt{sig\_seq 0 2 n}$$

$$\texttt{Rn nat\_seq n} \equiv \texttt{(nats ^^ S n)'}$$

The auxiliary streams `sig_seq` and `nat_seq` are defined as:

```
Fixpoint sig_seq (i k n : nat) : Stream Z :=
  match n with
  | O ⇒ #1 ⊕ Σ@{i,k} #1
  | S n ⇒ Σ@{i,k,S n} #1 ⊕ sig_seq i k n
  end.
Fixpoint nat_seq (n : nat) : Stream Z :=
  match n with
  | O ⇒ #1 ⊕ nats
  | S n ⇒ nats ⊙ (#1 ⊕ nats) ^^ S n ⊕ nat_seq n
  end.
```

The lemmas involving the above equalities are proven by induction. Equational reasoning is supported by Coq's `ring` tactic for solving ring equations.

```
Lemma Ssigmas_head_S i k n : head (Σ@{S i,k,n} #1) = 1.
Lemma Ssigmas_S_tail i k n : (Σ@{S i,k,n} #1)' ≡ sig_seq i k n.
Lemma nats_pow_head n : head (nats ^^ n) = 1.
Lemma nats_pow_tail n : (nats ^^ S n)' ≡ nat_seq n.
Lemma Rn_sig_seq_nat_seq n : Rn (sig_seq 0 2 n) (nat_seq n).
```

4.4 Case `Rn (Σ@{0,2,n} #1) (nats ⊙ (#1 ⊕ nats) ^^ n)`

In order to formalize the second case, we need to relate the heads and tails of `Σ@{0,2,n} #1` and `nats ⊙ (#1 ⊕ nats)^^ n`. This involves proving the equations below [13, Proposition 5.3-5.4]:

```
head (Σ@{0,2,n} #1) = 2 ^ n = head (nats ⊙ (#1 ⊕ nats) ^^ n)
    (Σ@{0,2,n} #1)' ≡ bins_sig_seq n 2 n
                  Rn  bins_seq n n ≡ (nats ⊙ (#1 ⊕ nats) ^^ n)'
```

The auxiliary streams `bins_sig_seq` and `bins_seq` are defined as:

```
Fixpoint bins_seq (n j : nat) : Stream Z :=
  match j with
  | O ⇒ #bins n n ⊙ (#1 ⊕ nats)
  | S j ⇒ #bins n (n - S j) ⊙ nats ⊙ (#1 ⊕ nats) ^^ S j ⊕ bins_seq n j
  end.
Fixpoint bins_sig_seq (n k j : nat) : Stream Z :=
  match j with
  | O ⇒ #bins n n ⊙ (#1 ⊕ Σ@{k - 2,k} #1)
  | S j ⇒ #bins n (n - S j) ⊙ Σ@{k - 2,k,S j} #1 ⊕ bins_sig_seq n k j
  end.
Fixpoint bins (n i : nat) : Z :=
  match i with O ⇒ 1 | S i ⇒ bin n (S i) + bins n i end.
```

In this code, `bin` n i denotes the binomial coefficient $\binom{n}{i}$. The series `bins` n i of binomial coefficients, denoted a_i^n by Niqui and Rutten, looks like:

$$a_i^n = \binom{n}{i} + \cdots + \binom{n}{1} + \binom{n}{0} = \binom{n}{i} + \cdots + \binom{n}{1} + 1.$$

The formal proofs in this section are surprisingly tricky and involve intricate generalizations of the induction hypothesis. These results thus form a nice example that informal proofs often hide too many details under the carpet; Niqui and Rutten just write that the results are "straightforward" or "proven by induction". Let us take a look at an example [13, Proposition 5.3]:

```
Lemma nats_nats_pow_head n : head (nats ⊙ (#1 ⊕ nats) ^^ n) = 2 ^ n.
Lemma nats_nats_pow_tail n : (nats ⊙ (#1 ⊕ nats) ^^ n)‘ ≡ bins_seq n n.
```

The proof of the lemma `nats_nats_pow_head` is a trivial induction proof. We use basic properties about the heads of the operations for element-wise addition, multiplication and repeated multiplication. The lemma `nats_nats_pow_tail` cannot be proven by a mere induction on `n`. For the inductive step we were in need of the following auxiliary result:

```
Lemma bins_seq_SS n : bins_seq (S n) (S n) ≡ (#2 ⊕ nats) ⊙ bins_seq n n.
```

The formal proof of this auxiliary result is tricky, and requires a subtle generalization of the induction hypothesis. The subtlety arises from the fact that the lemma concerns `bins_seq` with the same value (namely `S n`) shared by both arguments. However, `bins_seq` is defined recursively on its second argument, whereas the first argument remains constant throughout the recursion. Therefore, we had to generalize the lemma `bins_seq_SS` such that both arguments are independent. The Coq statement of the generalized lemma is as follows.

```
Lemma bins_seq_SS_help n j :
  (j < n)%nat →
  bins_seq (S n) (S j) ≡ (nats ⊕ #2) ⊙ bins_seq n j ⊕
                         #bins n (n - S j) ⊙ nats ⊙ (#1 ⊕ nats) ^^ S j.
```

The above lemma is proven by induction on `j`. The main lemma `bins_seq_SS` is then proven by case analysis on `n` using the generalized lemma.

The other results from Niqui and Rutten, in particular [13, Proposition 5.4], also require a variety of helping lemmas whose proofs involve intricate generalizations of the induction hypothesis.

5 Long and Salié's generalization

Long [10,11] and Salié [20] generalized Moessner's result to apply to the situation in which the initial sequence is not the sequence of successive integers $(1, 2, 3, \dots)$ but the arithmetic progression $(a, d+a, 2d+a, \dots)$. They showed that the final sequence obtained by the Moessner construction is $(a \cdot 1^{n-1}, (d+a) \cdot 2^{n-1}, (2d+a) \cdot 3^{n-1}, \dots)$. We show that these results are a corollary of the version of Moessner's Theorem proven in Section 4. This is a new proof: Niqui and Rutten did not have it in their paper.

Similar to Section 4, where we started with the constant stream $(1, 1, 1, \dots)$ instead of $(1, 2, 3, \dots)$, we will here start with $(a, d, d, \dots)$ instead of $(a, d+a, 2d+2, \dots)$. Clearly we have $\Sigma\,(a, d, d, \dots) \equiv (a, a+d, a+2d, \dots)$, and hence

$\Sigma^n_{n+1}\,(a, d, d, \dots) \equiv (a, a+d, a+2d, \dots)$ for any $n \geq 1$. We can formulate Long and Salié's generalization of Moessner's Theorem thus as follows.

```
Corollary Moessner_ext a d n :
  Σ@{1,2,n} (a ::: #d) ≡ Σ (a ::: #d) ⊙ nats ^^ n.
```

The key observation to prove this generalization is the following lemma.

```
Lemma Moessner_ext_help a d : Σ (a ::: #d) ≡ #d ⊙ nats ⊕ #(a - d).
```

This lemma is straightforward to prove by showing that the heads and tails of both sides are equal. This involves just basic equational reasoning. It is essential that we consider streams of *integers* instead of *naturals*, since we want to allow the subtraction operation on the right hand side.

In order to prove the actual theorem, namely `Moessner_ext`, we perform case analysis on n. For the case 0, the result trivially holds, and for the case $1 + m$ we use the following derivation:

$$\begin{aligned}
\Sigma^1_2 \cdots \Sigma^{(1+m)+1}_{(1+m)+2}\,(a ::: \overline{d}) &\equiv \Sigma^1_2 \cdots \Sigma^{m+1}_{m+2}\,\Sigma\,(a ::: \overline{d}) \\
&\equiv \Sigma^1_2 \cdots \Sigma^{m+1}_{m+2}\,(\overline{d} \odot \mathtt{nats} \oplus \overline{a - d}) && (2)\\
&\equiv \overline{d} \odot \Sigma^1_2 \cdots \Sigma^{m+1}_{m+2}\,\mathtt{nats} \oplus \overline{a - d} \odot \Sigma^1_2 \cdots \Sigma^{m+1}_{m+2}\,\overline{1} && (3)\\
&\equiv \overline{d} \odot \mathtt{nats}^{\langle 2+m \rangle} \oplus \overline{a - d} \odot \mathtt{nats}^{\langle 1+m \rangle} && (4)\\
&\equiv (\overline{d} \odot \mathtt{nats} \oplus \overline{a - d}) \odot \mathtt{nats}^{\langle 1+m \rangle} \\
&\equiv \Sigma\,(a ::: \overline{d}) \odot \mathtt{nats}^{\langle 1+m \rangle} && (5)
\end{aligned}$$

Step 2 and 5 use the Lemma `Moessner_ext_help`, step 3 uses the fact that addition and scalar multiplication distribute through the partial sum and drop operations, and step 4 uses Moessner's Theorem twice.

6 Conclusions

We have presented a Coq formalization of Niqui and Rutten's proof of Moessner's Theorem [13], as well as a new proof for the generalization of Moessner's Theorem by Long and Salié. We will summarize the lessons learned from doing coinductive proofs in Coq.

Although Coq's syntactic guard condition for corecursive definitions is often believed to be too weak, it was strong enough to formalize this non-trivial coinductive proof without many complications. Most definitions of stream operations as given by Niqui and Rutten had a straightforward translation into a corresponding Coq definition. For some operations (*e.g.* `nats` and Σ), we had to modify the definition slightly, but we could easily prove that our alternative definition indeed satisfies the equations as given by Niqui and Rutten.

The guard condition was also hardly of any concern for proving properties. We only proved basic properties by guarded corecursion, and thereafter we typically proved stream equalities using the coinduction principle, element-wise equality, or equational reasoning using previously proved algebraic properties.

Hence, in more involved proofs, there was never the issue of proofs being rejected because of Coq's guard condition.

A source of inconvenience is that Coq's Leibniz equality is not extensional, and we thus had to resort to bisimilarity to capture stream equality. However, using the setoid machinery we could easily circumvent this source of inconvenience without noticeable overhead. We still had to prove that all stream operations respect bisimilarity, but those proofs were trivial. Hence, it would be useful if there was automation to do such proofs.

The proof of Moessner's Theorem involved some reasoning about ring equations over streams. Coq's `ring` tactic turned out to be extremely valuable, because it could solve these equations fully automatically.

One thing worth remarking is that Coq's notation system with unicode characters made it possible to type the proofs in a close notation to the one used by Niqui and Rutten. While formalizing their proof, we did not find any factual errors in the results. The challenge was that despite the good presentation of all definitions and auxiliary results, most proofs were hidden under the carpet. The proofs of the main propositions [13, Proposition 5.1-5.4] were claimed to be trivialities whereas they turned out to be more involved than expected.

In this paper we have moreover given a concise and original proof of Long and Salié's generalization. Although formalization did not directly help us discovering this proof, it was definitely of indirect use. Namely, formalization makes it very attractable to make as much parts of the proof development reusable. This was indeed the key to discovering our proof of this generalization.

We conclude that formalizing coinductive proofs as Moessner's Theorem in Coq is feasible, and worth doing. Our Coq formalization constitutes of a small library on general operations and theory on streams (348 lines), a proof of Moessner's Theorem for the case $n = 1$ (34 lines), for the case $n = 2$ (57 lines), and the general case (319 lines, including Long and Salié's generalization). This total of 758 lines of Coq code (including white space), corresponds to approximately 7 and half pages of informal mathematical text, with many proofs omitted, and without Long and Salié's generalization.

Acknowledgments. The authors thank Dexter Kozen, Jan Rutten, Olivier Danvy, and the anonymous referees for useful comments and discussions. The last author learned from Frank many years ago that a good steak, a glass of excellent wine, and fantastic company can make the hardest of days seem distant and irrelevant in the grand scheme of things! Frank, we wish you the very best for the years to come!

References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in TCS. Springer, 2004.

2. M. Bickford, D. Kozen, and A. Silva. Formalizing Moessner's Theorem and generalizations in NUPRL. Available at `http://www.nuprl.org/documents/Moessner/`, 2013.
3. C. Clausen, O. Danvy, and M. Masuko. A characterization of Moessner's sieve. *Theoretical Compututer Science*, 546:244–256, 2014.
4. J. H. Conway and R. K. Guy. Moessner's magic. In *The Book of Numbers*, pages 63–65. Springer-Verlag, 1996.
5. Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 2013.
6. C. E. Giménez. *Un Calcul de Constructions Infinies et son Application à la vérification de systèmes communicants*. PhD thesis, L'École Normale Supérieure de Lyon, 1996.
7. R. Hinze. Scans and convolutions—a calculational proof of Moessner's theorem. In *IFL '08*, volume 5836 of *LNCS*, 2009.
8. R. Honsberger. *More Mathematical Morsels*. Dolciani Mathematical Expositions. Math. Assoc. Amer., 1991.
9. D. Kozen and A. Silva. On Moessner's Theorem. *The American Mathematical Monthly*, 120(2):131–139, 2013.
10. C. T. Long. On the Moessner theorem on integral powers. *The American Mathematical Monthly*, 73(8):846–851, 1966.
11. C. T. Long. Strike it out–add it up. *The Mathematical Gazette*, 66(438):273–277, 1982.
12. A. Moessner. Eine Bemerkung über die Potenzen der natürlichen Zahlen. *Sitzungsberichte der Bayerischen Akademie der Wissenschaften, Mathematischnaturwissenschaftliche Klasse 1952*, 29, 1951.
13. M. Niqui and J. J. M. M. Rutten. A proof of Moessner's theorem by coinduction. *Higher-Order and Symbolic Computation*, 24(3):191–206, 2011.
14. I. Paasche. Ein neuer Beweis des moessnerischen Satzes. *Sitzungsberichte der Bayerischen Akademie der Wissenschaften, Mathematischnaturwissenschaftliche Klasse 1952*, 1:1–5, 1953.
15. I. Paasche. Ein zahlentheoretische-logarithmischer Rechenstab. *Math. Naturwiss. Unterr.*, 6:26–28, 1953–54.
16. I. Paasche. Eine Verallgemeinerung des moessnerschen Satzes. *Compositio Mathematica*, 12:263–270, 1954.
17. O. Perron. Beweis des Moessnerschen Satzes. *Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse 1951*, 4:31–34, 1951.
18. J. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15:93–147, 2005.
19. J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Compututer Science*, 249(1):3–80, 2000.
20. H. Salié. Bemerkung zu einem Satz von Moessner. *Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse 1952*, 2:7–11, 1952.
21. M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.
22. M. Sozeau and N. Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.
23. P. Urbak. A Formal Study of Moessner's Sieve, 2015. MSc thesis, Aarhus University.