A Formal Study of Moessner's Sieve

Peter Urbak, 20081130

Master's Thesis, Computer Science
February 2015
Advisor: Olivier Danvy

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In this dissertation, we present a new characterization of Moessner's sieve that brings a range of new results with it. As such, we present a dual to Moessner's sieve that generates a sequence of so-called Moessner triangles, instead of a traditional sequence of successive powers, where each triangle is generated column by column, instead of row by row. Furthermore, we present a new characteristic function of Moessner's sieve that calculates the entries of the Moessner triangles generated by Moessner's sieve, without having to calculate the prefix of the sequence.

We prove Moessner's theorem adapted to our new dual sieve, called Moessner's idealized theorem, where we generalize the initial configuration from a sequence of natural numbers to a seed tuple containing just one non-zero entry. We discover a new property of Moessner's sieve that connects Moessner triangles of different rank, thus acting as a dual to the existing relation between Moessner triangles of different index, thereby suggesting the presence of a 2-dimensional grid of triangles, rather than the traditional 1-dimensional sequence of values.

We adapt Long's theorem to the dual sieve and obtain a simplified initial configuration of Long's theorem, consisting of a seed tuple of two non-zero entries. We conjecture a new generalization of Long's theorem that has a seed tuple of arbitrary entries for its initial configuration and connects Moessner's sieve with polynomial evaluation. Lastly, we approach the connection between Moessner's sieve and polynomial evaluation from an alternative perspective and prove an equivalence relation between the triangle creation procedures of Moessner's sieve and the repeated application of Horner's method for polynomial division.

All results presented in this dissertation have been formalized in the Coq proof assistant and proved using a minimal subset of the constructs and tactics available in the Coq language. As such, we demonstrate the potential of proof assistants to inspire new results while lowering the gap between programs (in computer science) and proofs (in mathematics).

Acknowledgments

First of all, I would like to express my gratitude towards Olivier Danvy for his invaluable guidance over the last six months along with his ability to see the potential of my abilities, even when I could not. His knowledge and good humor were a vital part in the process. I am also grateful to Ole Østerby for generously sharing his expertise about Horner's method and abstract algebra. Throughout, Markus Wüstenberg has been a wonderful office mate.

I also want to thank the designers and developers of the Coq proof assistant, Proof General, Emacs, and LaTeX: these tools have been instrumental here. Coq, in particular, makes it possible for computer scientists to actually do mathematics, as unexpected as that may be.

This dissertation is the punch line of six enriching years as student and also as employee in the Department of Computer Science at Aarhus University. It is also the punch line of an upbringing for which I am grateful to my family. Last but not least, I want to thank Sofie for her loving support and our many technical discussions.

Peter Urbak,
Aarhus, Friday 20*th* *February 2015.*

Contents

Prerequisites and notation

There are different rules for reading,
for thinking, and for talking.
Writing blends all three of them.

Mason Cooley

This dissertation abides with the traditional use of "we" in scientific writing.

Prerequisites

Since all our proof statements have been proved using Coq, we expect the reader to have a basic understanding of interactive theorem proving and, in particular, of writing proofs using the Coq proof assistant. In our proofs, we deliberately stay at an elementary level that restricts us to a limited set of proof tactics for equational reasoning, such as induction, coinduction, and case analysis.

Furthermore, we do not claim to be experts in the theories behind these proof techniques. Instead, we use the techniques as tools for proving non-trivial statements in the area of our discourse, specifically Moessner's theorem and Moessner's sieve.

Lastly, since equational reasoning lies at the core of this thesis, we expect the reader to be familiar with equational reasoning at a high-school level.

Notation

Throughout this dissertation, we liberally use the terms "algebra" and "calculus" to mean "a module of arithmetic-like operators on a type". As such, the reader can understand "algebraic" as "having to do with lists" and "coalgebraic" as "having to do with streams".

Furthermore, for the sake of keeping each chapter as self-contained as possible, we occasionally repeat points made in previous chapters, e.g., the relation between Moessner's sieve and Pascal's triangle, or we repeat existing definitions, e.g., the definition of Moessner's sieve.

Lastly, the source code of this dissertation can be found under the author's Github profile, `https://github.com/dragonwasrobot`.

Chapter 1

Introduction

I always write a good first line,
but I have trouble in writing the others.

MOLIÈRE

The goal of this chapter is to introduce the content of this dissertation by defining its foundation along with its contributions.

In this chapter, we first define Moessner's theorem and Moessner's sieve in Section 1.1, followed by stating our goals and contributions in Section 1.2. Lastly, we give an overview of the rest of the dissertation in Section 1.3.

1.1 Moessner's theorem and sieve

At the core of this thesis lies Moessner's theorem, conjectured by Alfred Moessner in 1951 [27], which describes how to generate the sequence of successive powers,

$$1^k, 2^k, 3^k, \dots, \tag{1.1}$$

where $k \in \mathbb{N}$ and $k \geq 2$, from the sequence of positive natural numbers,

$$1, 2, 3, \dots.$$

The procedure described by Moessner for generating the result sequence of successive powers, works by dropping every kth element of an initial sequence, where $2 \leq k \in \mathbb{N}$, and partially summing the remaining elements into a new sequence. This step of dropping and summing is iteratively repeated, where k is decreased by 1 for each iteration; the procedure stops when the value of k reaches 1. The simplest case, where $k = 2$, yields the procedure,

$$\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & \dots \\
1 & & 3 & & 5 & & \dots \\
1 & & 4 & & 9 & & \dots
\end{array}$$

where the values $(2, 4, 6, \dots)$ are dropped in the initial sequence, and the remaining values $(1, 3, 5, \dots)$ are partially summed to yield the sequence $(1, 4, 9, \dots)$. The rank k is decreased from 2 to 1 and the procedure stops. Here, we note that the resulting sequence is the sequence of squares,

$$1^2, 2^2, 3^2, \dots,$$

which is exactly the sequence of Formula 1.1,

$$1^k, 2^k, 3^k, \dots,$$

where $k = 2$. We refer to the above procedure of repeatedly dropping and partially summing element as 'Moessner's sieve', and refer to k as the rank of the sieve.

In the course of this dissertation, we cover the related work of Moessner's theorem and we build on it to uncover formalizations that shed new light on Moessner's sieve and Moessner's theorem – we even state a new conjecture related to an existing generalization of Moessner's theorem.

1.2 Goals and contributions

Since Moessner's sieve and Moessner's theorem are the subjects of this study, our goal is to give a characterization of Moessner's sieve while proving Moessner's theorem and some of its generalizations. As a result, we present the following contributions, which enabled several new results:

- A formalization of the dual of Moessner's sieve (abbreviated the 'dual sieve'), which generates a sequence of triangles, called 'Moessner triangles', instead of a result sequence of successive powers, where each triangle is created column by column, instead of row by row.

- The definition of a characteristic function of Moessner's sieve that calculates any given entry of a generated Moessner triangle, without calculating the prefix of the sieve, and which we also prove to be correct.

- A proof of Moessner's theorem adapted to the dual sieve, called Moessner's idealized theorem, and generalized to an initial configuration consisting of a seed tuple with a single nonzero entry of 1, rather than a sequence of positive natural numbers or an arithmetic progression.

- The introduction of a new property of Moessner's sieve that establishes a connection between Moessner triangles of different rank, thereby suggesting the presence of a grid of triangles, rather than simply a sequence.

- A proof of Long's theorem adapted to the dual sieve, called Long's idealized theorem, together with the statement of a conjecture that generalizes Long's idealized theorem from a seed tuple with two nonzero entries to a seed tuple with an arbitrary number of nonzero entries and connects the dual sieve to polynomial evaluation.

* An equivalence proof between the repeated application of Horner's method for polynomial division and the triangle creation procedure of the dual sieve, which further strengthens the relation between polynomial evaluation and Moessner's sieve.

1.3 Overview

The dissertation is structured as follows. In Chapter 2, we establish our foundation by going through the related work leading up to this thesis. Following this, we give an overview of the Coq proof assistant in Chapter 3, which we use as our tool of choice to reach our goals, and we elaborate on the set of features we use and the approach we take throughout the dissertation.

Having established the foundation on which we stand and the tools we use, we construct a list calculus and a stream calculus in Chapter 4 that serve as the framework with which we do our proofs. Furthermore, we also define a traditional version of Moessner's sieve working on streams that we use to introduce our dual in Chapter 5, which works on tuples and generates a sequence of Moessner triangles instead of a stream of successive powers.

Taking a step back, we discuss Pascal's triangle and the binomial coefficient function in Chapter 6 and we formalize the rotated Pascal's triangle and the rotated binomial coefficient function. In Chapter 7, we extend the definition of the rotated binomial coefficient function to a characteristic function of Moessner's sieve, by observing a relation between the Moessner triangles generated by Moessner's sieve and the binomial expansion.

Having defined the dual sieve and formalized its characteristic function, we state and prove Moessner's idealized theorem in Chapter 8 using these new constructs. As a consequence of our formalizations, we discover a new property that connects Moessner triangles of different rank, suggesting a grid of triangles, which we explore and formalize in Chapter 9.

In order to both test and explore the potential of the dual sieve as a generic formalization going beyond Moessner's theorem, we state and prove Long's theorem in Chapter 10 in an idealized form adapted to the dual sieve, while maintaining the spirit of Long's original theorem. Furthermore, we conjecture a generalization of Long's idealized theorem that connects Moessner's sieve to polynomial evaluation and extends the initial configuration of Moessner's sieve from a seed tuple of two nonzero entries to a seed tuple with an arbitrary number of nonzero entries.

In Chapter 11, we approach the connection between Moessner's sieve and polynomial evaluation from a different perspective, by proving an equivalence relation between the repeated application of Horner's method for polynomial division and the triangle creation procedure of the dual sieve.

Lastly, we conclude our findings in Chapter 12 and reflect on the process and future work.

This dissertation is supplemented with a glossary.

Chapter 2

Related Work

All good things come in pairs.

OLD CHINESE PROVERB

Fortune favors the prepared mind.

LOUIS PASTEUR

How would Lubitsch do it?

BILLY WILDER

The goal of this chapter is to put this dissertation, and Moessner's sieve, into a historical context by reviewing the work which has gone before it.

In this chapter, we repeat the definition of Moessner's sieve and Moessner's theorem, as described in Chapter 1, in order to keep the chapter self-contained.

The chapter is structured as follows. In Section 2.1, we review the algebraic approaches to Moessner's sieve starting with Moessner's original work, and touch upon some of the generalizations made, using inductive proof techniques. Furthermore, we also review articles that examine Moessner's sieve from alternative perspectives such as graph theory and circuit theory. As a dual to the algebraic approaches, we review the coalgebraic approaches to Moessner's sieve in Section 2.2, where we revisit many of the algebraic proofs, using coinductive proof techniques, while adding further generalizations to Moessner's theorem.

2.1 Algebraic approaches

This section examines the algebraic approaches to Moessner's sieve covered in the literature. When discussing algebraic approaches, we use the term 'finite sequence' to denote an ordered finite collection in a mathematical context, and the term 'list' to denote the computational representation of a finite sequence. Because of the prevalence of inductive proof techniques in the covered articles, we dedicate the first section to inductive contributions and the second section to non-inductive contributions.

2.1.1 Inductive contributions

In 1951, Alfred Moessner conjectured his now famous theorem [27], demonstrating how to generate the sequence of successive powers,

$$1^k, 2^k, 3^k, \ldots, \tag{2.1}$$

where $k \in \mathbb{N}$ and $k \geq 2$, from the sequence of positive natural numbers,

$$1, 2, 3, \ldots.$$

The procedure described by Moessner for generating the result sequence of successive powers, works by dropping every kth element of an initial sequence, where $2 \leq k \in \mathbb{N}$, and partially summing the remaining elements into a new sequence. This step of dropping and summing is repeated iteratively where k is decreased by 1 for each iteration; the procedure stops when the value of k reaches 1. The simplest case, where $k = 2$, yields the procedure,[1]

$$\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & \ldots \\
1 & & 3 & & 5 & & \ldots \\
1 & & 4 & & 9 & & \ldots
\end{array}$$

where the values $(2, 4, 6, \ldots)$ are dropped in the initial sequence, and the remaining values $(1, 3, 5, \ldots)$ are partially summed to yield the result sequence $(1, 4, 9, \ldots)$. The rank k is decreased from 2 to 1 and the procedure stops. Here, we note that the resulting sequence is the sequence of squares,

$$1^2, 2^2, 3^2, \ldots,$$

which is exactly the sequence of Formula 2.1,

$$1^k, 2^k, 3^k, \ldots, \tag{2.2}$$

when $k = 2$. We refer to the above procedure of repeatedly dropping and partially summing element as 'Moessner's sieve', and refer to k as the rank of the sieve.

[1] In all future examples we contract the dropping and partial summing into one step and mark the dropped elements by making them boldface.

Moessner's conjecture was quickly proved by Oskar Perron [34], less than a year after its initial publication. Following Perron's proof of Moessner's theorem, further generalizations were soon made by Ivan Paasche [32] and Hans Salié [40]. Paasche showed how incrementally increasing the gap between the elements dropped in Moessner's sieve,

$$\begin{array}{ccccccccccc}
\mathbf{1} & 2 & \mathbf{3} & 4 & 5 & \mathbf{6} & 7 & 8 & 9 & \mathbf{10} & \dots \\
 & \mathbf{2} & & 6 & \mathbf{11} & & 18 & 26 & \mathbf{35} & & \dots \\
 & & & \mathbf{6} & & & 24 & \mathbf{50} & & & \dots \\
 & & & & & & \mathbf{24} & & & & \dots
\end{array}$$

allowed him to obtain the sequence of factorials,

$$1!, 2!, 3!, \dots,$$

and super factorials,

$$1!!, 2!!, 3!!, \dots.$$

Salié proved another generalization of Moessner's theorem by applying Moessner's sieve on an arbitrary initial sequence,

$$a_1, a_2, a_3, \dots,$$

instead of applying it on the sequence of positive natural numbers.

In 1966, Moessner's sieve was picked up by Calvin T. Long [22], who observed that the triangles generated by Moessner's sieve,

$$\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & \dots \\
1 & 2 & 3 & \mathbf{4} & & 5 & 6 & 7 & \mathbf{8} & & 9 & 10 & 11 & \mathbf{12} & & \dots \\
1 & 3 & \mathbf{6} & & & 11 & 17 & \mathbf{24} & & & 33 & 43 & \mathbf{54} & & & \dots \\
1 & \mathbf{4} & & & & 15 & \mathbf{32} & & & & 65 & \mathbf{108} & & & & \dots \\
\mathbf{1} & & & & & \mathbf{16} & & & & & \mathbf{81} & & & & & \dots
\end{array} \tag{2.3}$$

which we refer to as Moessner triangles, are constructed in a similar way to Pascal's triangle,

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
\mathbf{1} & & \mathbf{4} & & \mathbf{6} & & \mathbf{4} & & \mathbf{1}
\end{array} \tag{2.4}$$

The similarity lies in the observation that each entry in Pascal's triangle is the sum of the two values immediately above it,

$$\begin{tikzcd}
1 \arrow[dr] & & 2 \arrow[dl] \\
 & 3 &
\end{tikzcd}$$

while each entry in a Moessner triangle is the sum of the value immediately above it (northern neighbor) and left of it (western neighbor),

$$\begin{tikzcd} & 2 \arrow[d] \\ 1 \arrow[r] & 3 \end{tikzcd}$$

suggesting an equivalence relation between the two functions generating the triangles. This connection is also emphasized by the first Moessner triangle, in Formula 2.3, having the same entries as Pascal's triangle, in Formula 2.4.

Long used this observation to prove a new generalization of Moessner's theorem involving the introduction of a generalized version of Pascal's triangle,

$$\begin{array}{ccccccccc}
 & & & & d_0 & & & & \\
 & & & a_1 & & d_1 & & & \\
 & & a_2 & & a_1+d_1 & & d_2 & & \\
 & a_3 & & a_2+a_1+d_1 & & a_1+d_1+d_2 & & d_3 & \\
a_4 & & a_3+a_2+a_1+d_1 & & a_2+2a_1+2d_1+d_2 & & a_1+d_1+d_2+d_3 & & d_4
\end{array}$$

starting from two arbitrary sequences, $(a_1, a_2, \ldots)$ and $(d_0, d_1, \ldots)$, instead of two sequences of 1s. Long then showed how applying Moessner's sieve to the arithmetic progression

$$a, a+d, a+2d, a+3d, \ldots, \tag{2.5}$$

yields the result sequence

$$a \cdot 1^{k-1}, (a+d) \cdot 2^{k-1}, (a+2d) \cdot 3^{k-1}, \ldots, \tag{2.6}$$

where $k-1$ is the number of iterations in Moessner's sieve. When letting $a = 1$ and $d = 1$, the initial sequence in Formula 2.5 corresponds to the positive natural numbers,

$$\begin{array}{ccccc}
1, & 1+1, & 1+2\cdot 1, & 1+3\cdot 1, & \ldots \\
1, & 2, & 3, & 4, & \ldots
\end{array}$$

and the result sequence, in Formula 2.6, becomes the sequence of successive powers,

$$\begin{array}{ccccc}
1\cdot 1^{k-1}, & (1+1)+2^{k-1}, & (1+2\cdot 1)\cdot 3^{k-1}, & (1+3\cdot 1)\cdot 4^{k-1}, & \ldots \\
1\cdot 1^{k-1}, & 2+2^{k-1}, & 3\cdot 3^{k-1}, & 4\cdot 4^{k-1}, & \ldots \\
1^k, & 2^k, & 3^k, & 4^k, & \ldots
\end{array}$$

yielding Moessner's theorem.

Besides the above generalization, Long also pointed out several other properties of Moessner's sieve [24,25,42], among these is a relation between the index values of the dropped elements in an initial sequence, and the values of the result sequence – we refer to the sequence of the dropped elements as the dropped sequence. Specifically, Long observed that dropping the elements whose index values correspond to the sequence,

$$k_1,\quad 2k_1+k_2,\quad 3k_1+2k_2+k_3,\quad 4k_1+3k_2+2k_3+k_4,\quad \ldots, \tag{2.7}$$

yields the sequence,

$$1^{k_1},\quad 2^{k_1}1^{k_2},\quad 3^{k_1}2^{k_2}1^{k_3},\quad 4^{k_1}3^{k_2}2^{k_3}1^{k_4},\quad \ldots, \tag{2.8}$$

where all k_i are natural numbers. Here, the sums in the dropped sequence of Formula 2.7 are mapped to products in the result sequence of Formula 2.8, and the products are mapped to exponents, i.e., the dropped element of the initial sequence having index value $2k_1 + k_2$ is mapped to the element $2^{k_1}1^{k_2}$ in the result sequence.

Finally, Long discussed the potential of Moessner's sieve to entice high school students to become interested in mathematics, by its simplicity and wonder [23].

2.1.2 Non-inductive contributions

In 1959, Jan van Yzeren [44] showed how to derive Moessner's theorem by dividing a polynomial with a binomial, using Horner's method [6, 16], and repeating the process for the resulting quotient part of the division, until reaching a polynomial of degree 0, i.e., a constant. Performing the repeated application of Horner's method on the polynomial,

$$f(x) = 1x^4 + 0x^3 + 0x^2 + 0x^1 + 0x^0,$$

yields the following Horner blocks,

$$\begin{array}{ccccc@{\qquad}ccccc@{\qquad}ccccc}
1 & 0 & 0 & 0 & 0 & 1 & 4 & 6 & 4 & 1 & 1 & 8 & 24 & 32 & 16 \\
 & 1 & 1 & 1 & 1 & & 1 & 5 & 11 & 15 & & 1 & 9 & 33 & 65 \\
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 5 & 11 & 15 & \mathbf{16} & 1 & 9 & 33 & 65 & \mathbf{81} \\
 & 1 & 2 & 3 & & & 1 & 6 & 17 & & & 1 & 10 & 43 & \\
1 & 2 & 3 & \mathbf{4} & & 1 & 6 & 17 & \mathbf{32} & & 1 & 10 & 43 & \mathbf{108} & \\
 & 1 & 3 & & & & 1 & 7 & & & & 1 & 11 & & \\
1 & 3 & \mathbf{6} & & & 1 & 7 & \mathbf{24} & & & 1 & 11 & \mathbf{54} & & \\
 & 1 & & & & & 1 & & & & & 1 & & & \\
\mathbf{1} & \mathbf{4} & & & & \mathbf{1} & \mathbf{8} & & & & \mathbf{1} & \mathbf{12} & & &
\end{array}$$

which have the same hypotenuse, highlighted in boldface, as the triangles generated in Formula 2.3, however in reversed order. Furthermore, if we remove every second row in the blocks above, along with the first row, we obtain a mirror image of the triangles in Formula 2.3,

$$\begin{array}{ccccc@{\quad}ccccc@{\quad}ccccc@{\quad}c}
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 5 & 11 & 15 & \mathbf{16} & 1 & 9 & 33 & 65 & \mathbf{81} & \ldots \\
1 & 2 & 3 & \mathbf{4} & & 1 & 6 & 17 & \mathbf{32} & & 1 & 10 & 43 & \mathbf{108} & & \ldots \\
1 & 3 & \mathbf{6} & & & 1 & 7 & \mathbf{24} & & & 1 & 11 & \mathbf{54} & & & \ldots \\
1 & \mathbf{4} & & & & 1 & \mathbf{8} & & & & 1 & \mathbf{12} & & & & \ldots \\
\mathbf{1} & & & & & \mathbf{1} & & & & & \mathbf{1} & & & & & \ldots
\end{array}$$

suggesting an equivalence relation between the two functions generating Moessner triangles and Horner blocks.

A different approach was taken by Karel A. Post [35], who used graph theory to model each entry in the sieve as a node and every addition as a directed edge. This structure allowed him to establish simple graph-theoretical proofs of Moessner's original theorem and the generalizations made by Paasche, Salié and Long.

Looking at Moessner's theorem from a circuit-theoretical perspective, Samadi et al. [41] contributed a more technical result describing how to compute the sequence of successive powers, by means of recursive and non-recursive multiplier-free circuit structures mimicking Moessner's sieve. This result shows that the ability of Moessner's sieve to calculate the powers of natural numbers without multiplication, expands its application beyond mathematical curiosity and into the more practical realm of circuits.

Lastly, Graham et al. [12] proved Moessner's theorem using power series, while more recent publications by Dexter Kozen and Alexandra Silva [18] used formal power series to prove Moessner's-, Paasche's- and Long's theorems. By using the theory of formal power series, Kozen and Silva proved a new generalization of Moessner's theorem which turned the mentioned proofs into simple corollaries. Besides pen and paper, these proofs have also been formalized, in cooperation with Mark Bickford, in the Nuprl proof development system [4, 5, 20].

2.2 Coalgebraic approaches

As a dual to the algebraic approach above, this section examines the coalgebraic approaches to Moessner's sieve. Here, we use the term 'infinite sequence' to denote an ordered infinite collection in a mathematical context, and the term 'stream' to denote the computational representation of an infinite sequence. Analogously to the previous section, we dedicate the first section to coinductive contributions and the second section to non-coinductive contributions.

2.2.1 Coinductive contributions

Extending on the theory of coalgebra [37], Jan Rutten developed a coinductive calculus of streams [38], founded on a small set of useful concepts to reason about streams: initial values, stream derivatives, stream differential equations, along with the coinduction definition and -proof principles. Using these concepts, Rutten demonstrated several applications of stream calculus in areas of discrete mathematics, analysis and combinatorics. Further work with the framework of stream calculus was done in collaboration with Milad Niqui [28, 31], where they studied various operations for partitioning, projecting and merging streams, and later used them to develop precise proofs for Moessner's theorem using coinductive proof techniques [29, 30]. Subsequently, the proofs by Niqui and Rutten of Moessner's theorem have been formalized in the Coq proof assistant [1] by Krebbers et al. [19]. Besides proving Moessner's theorem, their formalization made them able to create a foundation of proved properties that abstracted away the often brittle reasoning associated with Coq's guardedness condition for corecursive definitions. Furthermore, their formalization resulted in new proofs of the generalizations made by Long and Salié, which required only a minimal effort beyond their existing proof framework.

Other work in the Coq proof assistant has been done by Yves Bertot [2, 3] and Danvy et al. [7], where Bertot has formalized Eratosthenes' sieve and Danvy et al. have formalized Moessner's sieve. The article by Danvy et al. introduces the much needed name 'Moessner's sieve', in reference to Eratosthenes' sieve, to refer to the procedure of repeatedly dropping and partially summing sequences, that lies at the core of Moessner's theorem and its generalizations.

Besides coining the term 'Moessner's sieve', Danvy et al. also generalize the initial sequence of Moessner's theorem, by starting with a sequence consisting of a 1 followed by 0s,

$$1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad \ldots,$$

as opposed to the sequence of positive natural numbers,

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad \ldots.$$

This generalization provides a simpler basis, since the sequence of positive natural numbers can be obtained by applying Moessner's sieve on the sequence of 1 followed by 0s with rank $k = 3$,

$$\begin{array}{ccccccccccccc} 1 & 0 & \mathbf{0} & 0 & 0 & \mathbf{0} & 0 & 0 & \mathbf{0} & 0 & 0 & \mathbf{0} & \ldots \\ 1 & \mathbf{1} & & 1 & \mathbf{1} & & 1 & \mathbf{1} & & 1 & \mathbf{1} & & \ldots \\ \mathbf{1} & & & \mathbf{2} & & & \mathbf{3} & & & \mathbf{4} & & & \ldots \end{array} \tag{2.9}$$

Furthermore, they note that the values dropped by Moessner's sieve when applied on the sequence of positive natural numbers at rank k, k being a natural number, enumerate the successive monomials in the binomial expansion of $(1 + t)^k$,

$$(1+t)^k = \binom{k}{0} t^0 + \binom{k}{1} t^1 + \binom{k}{2} t^2 + \cdots + \binom{k}{k-1} t^{k-1} + \binom{k}{k} t^k,$$

t being the index of the triangle in Moessner's sieve, indexed from 1. As such, if we examine the three triangles in Formula 2.3, we note that the result sequence, $(1, 16, 81, \ldots)$, enumerate the values t^4, $(1^4, 2^4, 3^4, \ldots)$, which means that the first two hypotenuses of dropped elements should have the values:

$$\begin{aligned} (1+1)^4 &= 1 \cdot 1^0 + 4 \cdot 1^1 + 6 \cdot 1^2 + 4 \cdot 1^3 + 1^4 \\ &= 1 + 4 + 6 + 4 + 1, \end{aligned}$$

and,

$$\begin{aligned}(1+2)^4 &= 1 \cdot 2^0 + 4 \cdot 2^1 + 6 \cdot 2^2 + 4 \cdot 2^3 + 2^4 \\ &= 1 + 8 + 24 + 32 + 16.\end{aligned}$$

Comparing the terms of the two binomial expansions with the hypotenuses of the two triangles, we see that the dropped values are indeed enumerating the values of the monomials in the binomial expansions.

Danvy et al. also introduced a left inverse of Moessner's sieve, which given a result sequence and a rank returns the initial sequence, thus reversing the effect of applying Moessner's sieve:

$$\begin{array}{cccccccccccccc}
\mathbf{1} & & & & & & \mathbf{16} & & & & & & \dots \\
1 & \mathbf{4} & & & & & 15 & \mathbf{32} & & & & & \dots \\
1 & 3 & \mathbf{6} & & & & 11 & 17 & \mathbf{24} & & & & \dots \\
1 & 2 & 3 & \mathbf{4} & & & 5 & 6 & 7 & \mathbf{8} & & & \dots \\
1 & 1 & 1 & 1 & \mathbf{1} & & 1 & 1 & 1 & 1 & \mathbf{1} & & \dots \\
1 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & \dots
\end{array}$$

Lastly, Danvy et al. propose a generalization of Long's theorem [22] starting from a sequence consisting of an a followed by ds,

$$a \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad d \quad \dots,$$

instead of the arithmetic progression in Formula 2.5,

$$a \quad a+d \quad a+2d \quad a+3d \quad a+4d \quad a+5d \quad a+6d \quad \dots,$$

since the arithmetic progression can be obtained by applying Moessner's sieve on the sequence of an a followed by ds with $k = 2$ – analogously to the way the sequence of positive natural numbers could be obtained from the sequence of a 1 followed by 0s.

2.2.2 Non-coinductive contributions

Besides their coinductive contribution, Niqui and Rutten also explore stream circuits [39], they prove Moessner's theorem using equational reasoning [29] and generating functions [30], still within the context of stream calculus. Other alternatives to coinduction have been suggested by Ralf Hinze, who conducts an extended study of stream calculus [14] wherein he introduces the concept of unique solutions, which he uses to redevelop the theory of reccurrences, finite calculus and generating functions in a way that allows equational reasoning of coinductive data types, i.e., streams. Hinze further develops stream-generating functions by introducing scans and convolutions [15], which he then uses to prove Paasche's generalization of Moessner's theorem.

2.3 Summary

In this chapter, we have put this dissertation, and Moessner's sieve, into a historical context by reviewing the work which has gone before it.

To summarize the related work covered in this chapter, we first reviewed the algebraic approaches to Moessner's sieve, starting with Moessner's original conjecture followed by a selection of its generalizations – most notably those by Long. These generalizations demonstrated a range of significant properties of Moessner's sieve and Moessner's theorem, which were proved using inductive proof techniques. Besides the inductive contributions, we also reviewed a series of non-inductive contributions, among which were an article by van Yzeren that created a connection between Horner's method and Moessner's sieve.

As a dual to the first part of the chapter, we reviewed coalgebraic approaches to Moessner's sieve in the second part of the chapter. Here, coinductive contributions, made primarily by Rutten and Niqui, introduced and utilized stream calculus to give precise coinductive proofs of Moessner's theorem and several of the inductive generalizations. Besides the framework of stream calculus, the section also reviewed several contributions which used interactive theorem provers, most notably the Coq proof assistant, to reason about streams and Moessner's theorem in a computational setting. Of note is the article by Danvy et al., which coined the term 'Moessner's sieve' while showing several new properties of it. Hence, we use this article as the starting point of this dissertation. Lastly, we reviewed non-coinductive contributions which mainly focused on using generating functions and formal power series to reason about streams.

Reflecting on the literature covered in this chapter, we observe that Moessner's sieve is like the elephant being examined by a group of blind scientists; each group perceives the elephant from their technical perspective, yet it is the same elephant. Being blind scientists ourselves, but with the benefit of hindsight, we try not to cover the whole elephant but instead try to characterize the elephant. As such, we conclude that the articles by van Yzeren [44], Long [22,23], Niqui and Rutten [29], Danvy et al. [7], and Krebbers et al. [19] are the pieces of related work most relevant to our goals, since they span several generalizations of – and connections to – Moessner's sieve with a focus on machine-assisted theorem proving. In particular, we share the same perspective as Long of wanting to explore the intrinsic beauty of simple mathematical concepts, which we do on top of the foundation laid by Danvy et al.

Chapter 3

The Coq proof assistant

A stone cannot fly
Mother cannot fly
Ergo, Mother is a stone!

Ludvig Holberg, Erasmus Montanus

Ah, I see you have the machine that goes ping,
this is my favorite.

Monty Python, The Meaning of Life

We need heuristic reasoning
when we construct a strict proof,
as we need scaffolding
when we erect a building.

George Polya

The goal of this chapter is to give an overview of the Coq proof assistant and elaborate on the features we use and the approach we take.

The chapter is structured as follows. In Section 3.1, we give an overview of the Coq proof assistant by emphasizing its main features followed by a description of some of the most notable contributions made using the Coq proof assistant, in Section 3.2. Following this, we discuss the set of features of the Coq language that make up the core of the toolbox we use throughout the dissertation, in Section 3.3. Lastly, we discuss the specific approach we take when developing theorems in the Coq proof assistant, in Section 3.4.

3.1 An overview of the Coq proof assistant

The Coq proof assistant implements a dependently typed program specification and mathematical higher-level language called Gallina that is based on the calculus of inductive constructions [1,3], which itself is derived from the calculus of constructions, invented by Thierry Coquand and Gérard Huet [8]. As such, Coq is founded on the Curry-Howard correspondence [9], which captures the direct relation between mathematical proofs and computer programs (*proofs-as-programs*), and propositions and types (*propositions-as-types*). As a result, Coq verifies its proofs using a type-checking algorithm, which checks that a program (proof) has the correct type (proves the proposition). Furthermore, Coq provides a tactic language, ltac, that allows semi-automatic interactive theorem proving by constructing – potentially quite elaborate – tactics, which can be applied across different proofs.

3.2 The contributions of the Coq proof assistant

As mentioned in the previous section, Coq provides a tactic language which facilitates the automation of proofs. A notable example, which took great advantage of this feature, is the formalization and proof of the four-color theorem by Georges Gonthier's [11] in 2008. Gonthier's contribution is particularly significant because the four-color theorem is notorious for having a large number of cases that needs to be proved (approximately 10,000), which could only be done within a system that allows the automation of large parts of the proof.

However, automation is not Coq's only strength as shown by Xavier Leroy, who headed the development of the CompCert C compiler [21], which is written and verified in Coq and intended for compilation of mission-critical software written in the C language. While Coq does not itself provide I/O-capabilities, it can export its definitions to a set of general purpose ML-like languages, thus providing the ability to create verified software using Coq.

Recently, the emergence of homotopy type theory, a marriage between homotopy theory of mathematical descendent and type theory of theoretical computer science descendent, has sparked an increased interest in proof assistants like Coq and Agda, which has resulted in the implementation of the homotopy type theory in Coq and an accompanying book [43].

Lastly, with this dissertation, we show how the use of a proof assistant can help us shed new light on an existing subject in mathematics, specifically Moessner's sieve, by taking the approach of a computer scientist and explore the sieve using the computational features of the Coq proof assistant combined with an elementary approach to theorem proving relying on induction and equational reasoning.

3.3 The Coq language

In this section, we cover the main set of features of the Coq language that we use throughout the dissertation to prove our goals. As such, we discuss the notation mechanism, inductive types, coinductive types, equality, induction, and coinduction.

Notation

Coq provides an extendable notation mechanism that allows us to define our own notation for types and functions. Specifically, we use it to define familiar notation for constructing lists and streams, along with various infix notations for list and stream operators, and traditional notation for functions such as the power function and the binomial coefficient function. We show examples of this notation in the next sections.

Inductive types

Coq offers a command, `Inductive`, that allows us to define inductive types and propostions, such as the built-in inductive type over natural numbers,

```
Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

which has a nice accompanying notation that allows us to write natural numbers, such as `(S (S (S 0)))`, using conventional Arabic numerals, such as `3`. Furthermore, we can define functions, called `Fixpoints`, which perform computations on these inductive types by pattern matching on their constructors, e.g., `O` and `S` in case of `nat`, as can be seen by the following example implementation of the power function,

```
Fixpoint power (e b : nat) : nat :=
  match e with
    | O ⇒ 1
    | S e' ⇒ b * (b ^ e')
  end
  where "b ^ e" := (power e b).
```

where we return `1` if the exponent `e` was constructed by the base case, `O`, or we perform a recursive call if it was constructed by the inductive case, `S e'`. Note also the declaration of a new infix notation, `where "b ^ e" := (power e b)`, inside the definition of the function. Besides the `nat` type, we also use the `list` type extensively,

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

for which we extend the current notation,

```
Notation " [ ] " := nil : list_scope.
Notation " [ x ] " := (cons x nil) : list_scope.
Notation " [ x ; .. ; y ] " := (cons x .. (cons y nil) ..) : list_scope.
Infix "::" := cons (at level 60, right associativity) : list_scope.
```

such that we can construct lists as `[...]` and use `"::"` as infix notation for the `list` constructor cons.

Coinductive types

Just as we can define inductive types, so can we define coinductive types that allows us to define infinite data structures such as streams,

```
CoInductive Stream (A : Type) : Type :=
  Cons : A → Stream A → Stream A.
Notation "s ::: σ " := (Cons s σ ) (at level 60, right associativity).
```

which do not need to have a base case, and where we can define corecursive functions, called `CoFixpoints`, to perform computations on these infinite data structures. For example, we can define a function for making `Streams` like so,

```
CoFixpoint make_stream (f : nat → nat) (n : nat) : Stream nat :=
  n ::: make_stream f (f n).
```

starting from an initial value `n` and a function `f` which is repeatedly applied on the value `n` as the stream is constructed.

Equality

For all types, there exists a built-in equivalence relation, Leibniz equality $(=)$, which captures the minimal reflexive relation,

```
Inductive eq (A : Type) (x : A) : A → Prop :=
| eq_refl : x = x.
```

While this relation is sufficient for all inductively defined types, it falls short for coinductive types as it is too restrictive to be used for proving equivalence of two infinite data structures. In fact, if we want to capture equality between two streams constructed by two different procedures, we have to establish a weaker form of equality called bisimilarity, which is a coinductively defined predicate,

```
CoInductive bisimilarity (σ  τ : Stream nat) : Prop :=
  bisimilar : σ (0) = τ (0) →
              σ ' ∼ τ '→
              σ ∼ τ
where "σ  ∼ τ " := (bisimilarity σ τ ).
```

that relates two streams that are element-wise equal.[1] Furthermore, we can use the `Equivalence` predicate of the `Setoid` library to prove that bisimilarity is an equivalence,

[1]We use the notation $\sigma(0)$ to refer to the head of a stream, called the initial value, and we use σ' to refer to the tail of a stream, called the stream derivative.

```
Global Instance bisimilar_equivalence :
  Equivalence bisimilarity.
```

which allows us to perform basic substitution for equivalence proofs specified in terms of bisimilarity. However, in most cases we also need to show that all procedures that operate on streams respect bisimilarity, which is captured by the `Proper` proposition of the `Morphisms` library and looks like this,

```
Global Instance Cons_proper :
  Proper (eq ⟹ bisimilarity ⟹ bisimilarity) (@Cons nat).

Global Instance hd_proper :
  Proper (bisimilarity ⟹ eq) (@hd nat).

Global Instance tl_proper :
  Proper (bisimilarity ⟹ bisimilarity) (@tl nat).
```

for the basic selectors, `Cons`, `hd`, and `tl`, of the `Stream` type. The concepts of `bisimilarity`, `Equivalence`, and `Proper` are discussed further in Chapter 4, where we introduce our stream calculus.

Induction

Another important property of inductively defined types in Coq is the automatic definition of an induction principle for the defined type. For example, when defining the `nat` type we automatically define the type,

```
nat_rect : ∀ P : nat → Type,
  P 0 →
  (∀  (n : nat), P n → P (S n)) →
  (∀  (n : nat), P n).
```

which captures the induction principle of natural numbers, consisting of a base case `P 0`, and an inductive case `(∀ (n : nat), P n → P (S n))`, which needs to be proved in order to obtain the goal `(∀ (n : nat), P n)`. Furthermore, Coq provides a high-level convenience tactic, `induction`, as syntactic sugar on top of the `fix` tactic, which makes induction proofs pleasant to work with.

Coinduction

Just as in the case of equality, we do not get the same features for free as we do in the inductive case. As such, we do not get any automatic coinduction principle or convenience tactic when defining a coinductive type. Instead, we have to use the low-level `cofix` tactic which does not report any violation of the guardedness principle when constructing the proof of the bisimilarity proposition – it postpones the verification check until the `Qed` keyword. This can easily lead to a frustrating experience for any non-trivial proof, thus we adopt the newly developed third party library paco by Hur et al. [17] that provides an alternative coinduction tactic, `pcofix`, which adds safeguards to coinduction proofs such that any violations of the guardedness principle is reported immediately and disallowed. However, using this package requires us to first define a generating function for bisimilarity,

```
Inductive bisimilarity_gen bisimilarity :
  Stream nat → Stream nat → Prop :=
  | _bisimilarity_gen :
      ∀ (σ  τ : Stream nat)
        (R_initial_values : σ (0) = τ (0))
        (R_stream_derivatives : (bisimilarity (σ ′) (τ ′) : Prop)),
        bisimilarity_gen bisimilarity σ τ .
```

which exhibit a similar structure to the coinductive `bisimilarity` relation we defined above. If we then apply our generating function `bisimilarity_gen` to the library function `paco2`,

```
Definition bisimilarity (σ  τ : Stream nat) : Prop :=
  paco2 bisimilarity_gen bot2 σ τ .
Infix "∼ " := bisimilarity (at level 70, no associativity).
```

we obtain an alternative version of the bisimilarity relation. With this relation defined, we can use the new `pcofix` tactic as a safe version of the traditional `cofix` tactic, without further issues.

Since, the paco library is solving a mainly technical issue in the context of our thesis, we will not go into further details with the theory behind it. Hence, when we define bisimilarity again in Chapter 4, we introduce the traditional version of bisimilarity defined as a `CoInductive` relation.

Having covered the set of features that define the subset of Coq we use in this dissertation, we move on to discuss our approach to constructing proofs in the Coq proof assistant.

3.4 An elementary approach to interactive theorem proving

In this dissertation, we take an elementary approach to interactive theorem proving in Coq, by using a minimal set of tactics and proofs. Specifically, we rely almost exclusively on equational reasoning combined with induction and coinduction, which makes our proofs accessible to people with a basic understanding of mathematics and Coq, while still managing to prove new mathematical results.

Furthermore, we do not rely on heavy automation or composite tactics like `simpl` and `auto`, but instead take an atomic approach where every step is clearly specified, thus providing a thorough documentation of how every proof is carried out. This has the added advantage that the Coq scripts end up spelling out every immediate dependency between proofs, which allows us to write a Python script that constructs dependency graphs for every Coq script, by parsing them. As a result, we present a dependency graph at the end of each chapter that reflects the scripts associated with the specific chapter. As such, we model every proof as a round node having a directed edge to every proof that depends on it, and fill every node representing a `Theorem` with purple to emphasize the most significant proofs of every chapter.

Lastly, for every `Definition`, `Fixpoint`, and `CoFixpoint` in our proof scripts, we define so-called unfolding lemmas that describe each possible case of the body of a function. For example, in the case of the `power` function defined previously, we have the following two unfolding lemmas,

```
Lemma unfold_power_base_case :
  ∀ b : nat,
    b ^ 0 = 1.
```

and

```
Lemma unfold_power_induction_case :
  ∀ b e' : nat,
    b ^ (S e') = b * (b ^ e').
```

corresponding to the cases `e = 0` and `e = S e'`. As a result, we actively avoid the use of the `unfold` tactic outside of our unfolding lemmas, which turns our unfolding lemmas into an interface that allows us to redefine `power` as long as it satisfies the two unfolding lemmas.

Thus, we have now given an overview of the Coq proof assistant, emphazied some of the major contributions made with it, put our work into a Coq context, and described the features and tools we use to obtain our results along with the approach we use for obtaining these.

3.5 Summary

In this chapter we have given an overview of the Coq proof assistant and elaborated on the features we use and the approach we take for interactive theorem proving.

Specifically, we have described the underlying logic of the Coq proof assistant and some of the major contributions made using it. Furthermore, we have discussed the notation, the types (inductive and coinductive), and a few advanced features of the Coq ecosystem we use to establish the foundation on which we build our proofs. Lastly, we have described the elementary approach we take when working with Coq, which emphasizes the potential of Coq to facilitate the structural approach of computer scientists to prove new theorems in areas traditionally reserved for mathematicians.

Chapter 4

Lists and streams

A programmer must be able
to express himself extremely well,
both in a natural language
and in the formal systems.

Edsger W. Dijkstra, 1973 (EWD361)

In this chapter, we establish a list calculus and a stream calculus that we use throughout the dissertation as a solid foundation on which to construct many of our proofs. Furthermore, we introduce procedures for generating the traditional version of Moessner's sieve working on streams.

The chapter is structured as follows. In Section 4.1 we define our list calculus consisting of a list type including a set of selectors, constructors and operators. Dually, we define our stream calculus in Section 4.2 along with a version of Moessner's sieve working on streams in Section 4.3.

4.1 List calculus

In this section, we define a list calculus composed of a list type together with a set of list selectors, constructors and operators that acts as the foundation for much of our inductive reasoning on Moessner's sieve. First, we define the basics of our calculus, i.e., the `list` type and its selectors and constructors, followed by a range of list operators such as `list_map` and `list_zip`.

4.1.1 List basics

In order to construct our list calculus, we start by defining exactly what we mean by a list. As such, we define a list to either be the empty list or an element followed by a list. Formalizing this description, we get the inductive type `list`,

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

whose base case is the empty list, denoted `nil`, and whose inductive case is an element followed by a `list`, captured by the constructor `cons`. Furthermore, we adopt the notation of writing a list as a set of brackets `[...]`,

```
Notation " [ ] " := nil : list_scope.
Notation " [ x ] " := (cons x nil) : list_scope.
Notation " [ x ; .. ; y ] " := (cons x .. (cons y nil) ..) : list_scope.
Infix "::" := cons (at level 60, right associativity) : list_scope.
```

along with the infix notation for `cons`, `'::'`, which provides a more readable notation for specifying lists.

Now that we can construct lists, we proceed by defining selectors which destruct elements of the `list` type by pattern matching on their structure. As a dual to `cons`, which takes an element and a `list` and constructs a new `list`, we define the selector `hd`,

```
Definition hd {A : Type} (d : A) (xs : list A) : A :=
  match xs with
    | [] ⇒ d
    | x :: _ ⇒ x
  end.
```

which takes a `list`, `xs`, and returns its head, if it exists, and the selector `tl`,

```
Definition tl {A : Type} (xs : list A) : list A :=
  match xs with
    | [] ⇒ []
    | _ :: xs' ⇒ xs'
  end.
```

which takes a `list`, `xs`, and returns its tail. We are now able to get the *n*th element of a `list` by applying `tl` *n* times on that `list` followed by an application of `hd`. However, such a definition is a bit inconvenient in a Coq context, so instead we have to merge the logic of the two functions, `hd` and `tl`, into the following selector,

```
Fixpoint nth {A : Type} (n : nat) (xs: list A) (d : A) : A :=
  match n, xs with
    | O, x :: xs' ⇒ x
    | O, [] ⇒ d
    | S n', [] ⇒ d
    | S n', x :: xs ⇒ nth n' xs' d
  end.
```

which takes an element index, `n`, a default value, `d`, and a `list`, `xs`, and returns the *n*th element of `xs`, if it exists.

Just as we can take the *n*th element, we can also define a method which returns the last element of a given `list`,

```
Fixpoint last {A : Type} (xs: list A) (d : A) : A :=
  match xs with
    | [] ⇒ d
    | [x] ⇒ x
    | x :: xs ⇒ last xs d
  end.
```

and similarly we can remove the last element of a `list`, xs,

```
Fixpoint removelast {A : Type} (xs : list A) : list A :=
  match xs with
    | [] ⇒ []
    | [x] ⇒ []
    | x :: xs ⇒ x :: removelast xs
  end.
```

by creating a new `list` containing all but the last element. Furthermore, we can also count the length of a `list`, xs,

```
Fixpoint length {A : Type} (xs : list A) : nat :=
  match xs with
   | [] ⇒ O
   | _ :: xs' ⇒ S (length xs')
  end.
```

and append one `list`, ys, to another `list`, xs,

```
Fixpoint app {A : Type} (xs ys : list A) : list A :=
  match xs with
    | [] ⇒ ys
    | x :: xs' ⇒ x :: app xs' ys
  end.
Infix "++ " := app (right associativity, at level 60) : list_scope.
```

where we use `'++'` to denote infix appending of lists. Using app, we can also reverse a `list`, xs,

```
Fixpoint rev {A : Type} (xs : list A) : list A :=
  match xs with
    | [] ⇒ []
    | x :: xs' ⇒ rev xs' ++ [x]
  end.
```

and prove that the first element of a reversed list is equal to the last element,

```
Lemma nth_rev_eq_last :
  ∀ (xs : list nat) (d : nat),
    nth 0 (rev xs) d =
    last xs d.
```

which we do by induction on the structure of the `list`, xs, i.e., by proving that the property holds for the base case of the empty list, `[]`, and for the inductive case of an element followed by a list, `x :: xs'`.

Lastly, we can define a procedure for constructing a `list` given a set of arguments, rather than having to construct the list by hand using `cons`. Specifically, we define a procedure, `make_list`, which takes a length, `n`, a seed value, `i`, and a progress function, `f`, for which it then constructs the `list` by repeatedly applying `f` on `i`, for each recursive call made to the procedure. We can translate this description into the following `Fixpoint`,

```
Fixpoint make_list (n i : nat) (f : nat → nat) : list nat :=
  match n with
    | 0 ⇒ []
    | S n' ⇒ i :: (make_list n' (f i) f)
  end.
```

which we use to define procedures for constructing a list of constants,

```
Definition list_constant (n c : nat) : list nat :=
  make_list n c (λ x : nat ⇒ x).
```

and a list of successive values,

```
Definition list_successor (n i : nat) : list nat :=
  make_list n i S.
```

allowing us to prove properties about `make_list` that we then get for free for all procedures defined in terms of `make_list`.

Having covered the `list` type and its selectors and constructors, we proceed by introducing a series of list procedures, which we call list operators, that manipulate all elements of a `list`, such as the familiar map and zip procedures.

4.1.2 List operators

The first list operator we define is `list_map`,

```
Fixpoint list_map (f : nat → nat) (xs : list nat) : list nat :=
  match xs with
    | [] ⇒ xs
    | x :: xs' ⇒ (f x) :: (list_map f xs')
  end.
```

which applies a function, `f`, on all elements of a `list`, `xs`, thus providing a generic base on which to define a range of procedures, as in the case of `make_list`. As a result, we can define scalar multiplication for lists in terms of `list_map`,

```
Definition list_scalar_multiplication (k : nat)
         (xs : list nat) : list nat :=
  list_map (mult k) xs.
Notation "k ⊗ xs" := (list_scalar_multiplication k xs)
                      (at level 40, left associativity).
```

such that `f` is the partial application `(mult k)`, and `k` is the natural number we multiply with every element of the list, `xs`.

Going from applying a function on one list, using `list_map`, we move on to define a procedure which merges two lists using a function, `list_zip`,

```
Fixpoint list_zip (f : nat → nat → nat)
       (xs ys : list nat) : list nat :=
  match xs, ys with
    | xs, [] ⇒ xs
    | [], ys ⇒ ys
    | x :: xs', y :: ys' ⇒ (f x y) :: (list_zip f xs' ys')
  end.
```

allowing us to define common procedures such as element-wise addition,

```
Definition list_sum (xs ys : list nat) : list nat :=
  list_zip plus xs ys.
Infix "⊕ " := list_sum (at level 50, left associativity).
```

and element-wise multiplication,

```
Definition list_product (xs ys : list nat) : list nat :=
  list_zip mult xs ys.
Infix "⊙ " := list_product (at level 40, left associativity).
```

for a pair of lists, `xs` and `ys`.

Lastly, we define two operators for partially summing a `list`; one with an accumulator, `a`,

```
Fixpoint list_partial_sums_acc (a : nat) (xs : list nat) : list nat :=
  match xs with
    | [] ⇒ []
    | x :: xs' ⇒ (x + a) :: (list_partial_sums_acc (x + a) xs')
  end.
```

and one initialized to `0`,

```
Definition list_partial_sums (xs : list nat) : list nat :=
  list_partial_sums_acc 0 xs.
```

defined in terms of the first.

This concludes the introduction of our list calculus, which is the first piece of the foundation we need in order to do proper reasoning about Moessner's sieve and its dual in the later chapters. As a dual to the list calculus, our next step is to define an equivalent stream calculus, which becomes the second large piece of the foundation on which we build our reasoning.

4.2 Stream calculus

Analogously to the previous section, we now define a stream calculus composed of a stream type together with a set of stream selectors, constructors and operators that constitute the second part of our proof foundation of a list and stream calculus.

In the first section, we define the basics of streams consisting of the `Stream` type, its selectors and constructors. However, before we define our stream operators, we first have to define stream equality, called bisimilarity, as streams are a coinductive data type that requires us to define our own measure of equality to reason about them. Furthermore, we also introduce the coinduction principle which allows us to prove properties over streams by induction on their element indices. Lastly, we introduce analogous stream operators to the operators we have already introduced for lists.

4.2.1 Stream basics

Just as we started the introduction of our list calculus by defining the `list` type, so do we start our stream calculus by defining the `Stream` type. We define a stream to be a coinductive type which has no base case, but only a coinductive case where a stream is constructed by adding an element onto an existing stream. This description yields the following formalization,

```
CoInductive Stream (A : Type) : Type :=
  Cons : A → Stream A → Stream A.
Notation "s ::: σ " := (Cons s σ ) (at level 60, right associativity).
```

where `Cons` is the single constructor of a `Stream`. Analogously to the `list` case, we define an infix notation for `Cons`, '`:::`', which provides a more convenient notation for writing `Stream`s.

In the same way as we defined selectors for getting the head and tail of a `list`, we now define selectors for destructing `Stream`s. As such, we introduce the selectors `hd` and `tl`,

```
Definition hd {A : Type} (σ  : Stream A) :=
  match σ with
    |  s ::: _ ⇒ s
  end.

Definition tl {A : Type} (σ  : Stream A) :=
  match σ with
    | _ ::: σ ' ⇒ σ '
  end.
```

which return the head and tail of a `Stream`. Being influenced by the stream calculus defined by Rutten [38], we adopt its notation and vocabulary by referring to the head of a `Stream` as its initial value, written $\sigma(0)$, and the tail as its stream derivative, written σ',

```
Notation "σ  (0)" := (hd σ ) (at level 8, left associativity).

Notation "σ  '" := (tl σ ) (at level 8, left associativity).
```

Using this notation, we can write the decomposition of a `Stream` as,

```
Lemma decompose_Stream :
  ∀ (σ  : Stream nat),
    σ = σ (0) ::: σ '.
```

which provides a clear syntax for unfolding a Stream into its initial value and stream derivative.

Similar to the list type, we also want to be able to get the nth element of a Stream. However, this time we first define a procedure, Str_nth_tl, that returns the nth tail, or stream derivative, of a Stream,

```
Fixpoint Str_nth_tl {A : Type} (n : nat) (σ : Stream A) : Stream A :=
  match n with
  | O ⇒ σ
  | S n' ⇒ Str_nth_tl n' σ'
  end.
```

with which we define Str_nth as the initial value of Str_nth_tl,

```
Definition Str_nth {A : Type} (n : nat) (σ : Stream A) : A :=
  (Str_nth_tl n σ)(0).
```

Lastly, we want to define the Str_prefix procedure, which establishes a connection between our list calculus and stream calculus, by returning the prefix of a Stream as a list,

```
Fixpoint Str_prefix (n : nat) (σ : Stream nat) : list nat :=
  match n with
    | 0 ⇒ []
    | S n' ⇒ σ(0) :: (Str_prefix n' σ')
  end.
```

The three selectors defined above, Str_nth_tl, Str_nth, and Str_prefix, form a powerful trio as they provide three different ways of reasoning about Streams – a point we make again in Section 4.2.3, when covering the coinduction principle.

Completely analogous to the list constructors of the previous section, we define a corecursive function for creating a Stream based on a seed value, n, and a progress function, f,

```
CoFixpoint make_stream (f : nat → nat) (n : nat) : Stream nat :=
  n ::: make_stream f (f n).
```

With this, we define a constant stream,

```
Definition stream_constant (c : nat) : Stream nat :=
  make_stream (λ x : nat ⇒ x) c.
Notation "# c" := (stream_constant c) (at level 4, left associativity).
```

where we use '#' as an abbreviation for stream_constant, and a successor procedure,

```
Definition stream_successor (i : nat) : Stream nat :=
  make_stream S i.
```

that is dual to list_successor.

Having defined the basics of the Stream type, we take a slight detour to discuss stream equality and the coinduction principle before introducing our set of stream operators.

4.2.2 Stream equality

While we are able to prove lemmas such as `decompose_Stream`, which states that a `Stream` is equal to the composition of its initial value and stream derivative, we are not immediately able to prove that two streams that are created by two different procedures, yet behave in the same way, are equal. The reason for this is that Leibniz equality $(=)$ captures a structural equivalence, e.g., we can observe that two inductively defined lists are structurally equivalent by traversing them and checking that each of their members are equal, which we cannot do for coinductively defined streams as they are infinite. Instead, we can observe that two streams produce the same elements whenever we ask for the next one. Thus, we can define a slightly weaker measure of equivalence, named bisimilarity $(\sim)$, which captures the behavioral equivalence of two streams. As such we define two streams, σ and τ, to be bisimilar if their initial values, $\sigma(0)$ and $\tau(0)$, are Leibniz equivalent, as we expect these to be `nats`, and their stream derivatives, σ' and τ', are bisimilar,

```
CoInductive bisimilarity (σ  τ : Stream nat) : Prop :=
  bisimilar : σ (0) = τ (0) →
              σ '∼ τ '→
              σ ∼ τ
where "σ  ∼ τ " := (bisimilarity σ τ ).
```

Just as we can use induction to prove that two lists are equal, by showing that their base cases and their inductive cases are Leibniz equivalent, so can we use coinduction to prove that two streams are bisimilar, by showing that their initial values are Leibniz equivalent and their stream derivatives are bisimilar.

Taking a step back, we can even define what we mean by a bisimulation,

```
Definition bisimulation (R : relation (Stream nat)) : Prop :=
 ∀ (σ  τ : Stream nat),
   R σ τ → σ (0) = τ (0) ∧ R σ 'τ '.
```

which is a relation, `R`, over two streams, σ and τ, for which their initial values are Leibniz equivalent, $\sigma(0) = \tau(0)$, and their stream derivatives are inhabitants of the relation, `R` σ' τ'. Consequently, we can now prove that our `bisimilarity` relation is indeed a `bisimulation`,

```
Lemma bisimilarity_is_a_bisimulation :
  bisimulation bisimilarity.
```

by destructing the definition of `bisimilarity` from which the `bisimulation` implication,

```
    ∀ σ τ : Stream nat,
      σ ∼ τ → σ (0) = τ (0) ∧ σ '∼ τ '
```

follows trivially. Lastly, we can prove that for any relation `R`, which is itself a `bisimulation`, it follows that two streams which inhabit the relation `R` are also bisimilar,

```
Lemma bisimulation_implies_bisimilarity :
  ∀ (R : relation (Stream nat)),
    bisimulation R → ∀ (σ  τ : Stream nat), R σ τ → σ ∼ τ .
```

The implication can be proved using coinduction, and by destructing the hypothesis (`bisimulation R`). Lastly, we can combine the two lemmas above and prove the bisimulation principle,

```
Theorem bisimulation_principle :
  ∀ (σ  τ : Stream nat),
    σ ∼ τ ↔ ∃ (R : relation (Stream nat)), bisimulation R ∧ R σ τ .
```

which states that two streams, σ and τ, are bisimilar if and only if there exists a relation R, such that R is a bisimulation in which σ and τ are inhabitants.

While Leibniz equality allows us to substitute y for x if we have proved $x = y$, we are not so fortunate when it comes to our newly defined `bisimilarity` relation. In order to use it as we would with Leibniz equality, we have to prove that the bisimilarity relation is indeed an `Equivalence`,

```
Global Instance bisimilar_equivalence :
  Equivalence bisimilarity.
```

which is done by proving that `bisimilarity` is reflexive, symmetric, and transitive,

```
Theorem bisimilarity_is_reflexive :
  ∀ (σ  : Stream nat),
    σ ∼ σ .

Theorem bisimilarity_is_symmetric :
  ∀ (σ  τ : Stream nat),
    σ ∼ τ → τ ∼ σ .

Theorem bisimilarity_is_transitive :
  ∀ (σ  τ ρ : Stream nat),
    σ ∼ τ → τ ∼ ρ → σ ∼ ρ .
```

Each of the three theorems is proved by coinduction and destructing the bisimilarity hypothesis into equivalence proofs of the initial values and stream derivatives.

Besides proving that our bisimilarity relation is indeed an equivalence, we also have to prove that all procedures which operate on streams, respect `bisimilarity`. A stream operator respects `bisimilarity` if two streams that are bisimilar before an application of the stream operator, are also bisimilar afterwards. For example, we state that `Cons`, `hd`, and `tl` all respect bisimilarity,

```
Global Instance Cons_proper :
  Proper (eq ⟹ bisimilarity ⟹ bisimilarity) (@Cons nat).

Global Instance hd_proper :
  Proper (bisimilarity ⟹ eq) (@hd nat).
```

```
Global Instance tl_proper :
  Proper (bisimilarity ⟹ bisimilarity) (@tl nat).
```

using the Proper predicate, and prove it using coinduction. Going into further details about respectfulness proofs are outside the scope of this dissertation, and hence we do not mention them beyond this paragraph.

Having introduced a Stream equality measure, bisimilarity, and proved that it is a bisimulation and an Equivalence, which allows us to rewrite with the proofs of bisimilarity between two streams, we prove the coinduction principle as our next step, which states a relation between bisimilarity and element-wise equality.

4.2.3 The coinduction principle

We now prove the coinduction principle which states that if two streams are bisimilar then they are also element-wise equal,

```
Theorem bisimilarity_iff_Str_nth :
  ∀ (σ  τ : Stream nat),
    σ ∼ τ ↔ (∀  (n : nat), Str_nth n σ = Str_nth n τ ).
```

where element-wise equality is represented by Str_nth. Proving the implication from left to right,

```
Lemma bisimilarity_implies_Str_nth :
  ∀ (n : nat) (σ  τ : Stream nat),
    σ ∼ τ → Str_nth n σ = Str_nth n τ .
```

is done by induction on the element index, n, and destructing the hypothesis, while the implication from right to left,

```
Lemma Str_nth_implies_bisimilarity :
  ∀ (σ  τ : Stream nat),
    (∀  (n : nat), Str_nth n σ = Str_nth n τ ) → σ ∼ τ .
```

is done by coinduction and applying the hypothesis of element-wise equality.

Besides proving that two streams are bisimilar if and only if every element of the two streams are equal, we can also prove that two streams are bisimilar if and only if all their tails, or stream derivatives, are bisimilar,

```
Theorem bisimilarity_iff_Str_nth_tl :
  ∀ (σ  τ : Stream nat),
    σ ∼ τ ↔ (∀  (n : nat), Str_nth_tl n σ ∼ Str_nth_tl n τ ).
```

Since both sides of the

proposition consists of bisimilarities,

```
Lemma bisimilarity_implies_Str_nth_tl :
  ∀ (n : nat) (σ  τ : Stream nat),
    σ ∼ τ → Str_nth_tl n σ ∼ Str_nth_tl n τ .

Lemma Str_nth_tl_implies_bisimilarity :
  ∀ (σ  τ : Stream nat),
    (∀  (n : nat), Str_nth_tl n σ ∼ Str_nth_tl n τ ) → σ ∼ τ .
```

we can prove them both using coinduction. However, for the first case we also rely on `bisimilarity_implies_Str_nth` to prove the equality of the two initial values and likewise rely on `tl_nth_tl`,

```
Lemma tl_nth_tl :
  ∀ (n : nat) (σ  : Stream nat),
    (Str_nth_tl n σ )′ = Str_nth_tl n σ ′.
```

when proving that the two stream derivatives are bisimilar.

Lastly, we can prove a relation which connects `list` equality, represented by `Str_prefix`, and `Stream` equality,

```
Theorem bisimilarity_iff_Str_prefix :
  ∀ (σ  τ : Stream nat),
    σ ∼  τ ↔ (∀  (n : nat), Str_prefix n σ = Str_prefix n τ ).
```

thus providing a powerful tool for which we can move between an inductive and coinductive world. Again, the implication from `Streams` to `lists` is proved using induction on the prefix length, `n`, and the implication from `lists` to `Streams` is proved by coinduction.

Having established the above set of equivalence proofs between the different approaches to proving stream equality, `Str_nth_tl`, `Str_nth` and `Str_-prefix`, we are now able to choose the setting that suits our method of attack the best, e.g., proving a statement with element-wise equality and then use the coinduction principle, `bisimilarity_iff_Str_nth`, to obtain the equivalent bisimilarity proof. Furthermore, by being able to go from coinduction to induction we get all the nice properties of the `induction` tactic and Leibniz equality, which we can otherwise only dream of when using `cofix` and bisimilarity.

Now that we have covered stream equality and the coinduction principle, we are ready to introduce our set of stream operators.

4.2.4 Stream operators

As a dual to the list operators introduced in Section 4.1.2, we now define similar operators for streams.

Just as we have defined `list_map` to apply a function, `f`, on every element of a `list`, `xs`, so can we define `stream_map` to apply a function, `f`, on every element of a `Stream`, σ,

```
CoFixpoint stream_map (f : nat → nat)
          (σ  : Stream nat) : Stream nat :=
  f σ (0) ::: stream_map f σ ′.
```

Similarly, if we let `f` be the partial application `(mult k)`, where `k` is a natural number, we obtain scalar multiplication for streams,

```
Definition stream_scalar_multiplication (k : nat)
          (σ  : Stream nat) : Stream nat :=
  stream_map (mult k) σ .
```

```
Notation "k ⊗ σ " := (stream_scalar_multiplication k σ )
                    (at level 40, left associativity).
```

Furthermore, we also define a zip procedure for streams,

```
CoFixpoint stream_zip (f : nat → nat → nat)
         (σ  τ : Stream nat) : Stream nat :=
  f σ (0) τ (0) ::: stream_zip f σ′τ′.
```

which merges two streams, σ and τ, using a function, `f`, that takes two natural numbers. Letting `f` be the function `plus` we get the `stream_sum` procedure,

```
Definition stream_sum (σ  τ : Stream nat) : Stream nat :=
  stream_zip plus σ τ .
Infix "⊕ " := stream_sum (at level 50, left associativity).
```

and if we let `f` be `mult` we get the `stream_product` procedure,

```
Definition stream_product (σ  τ : Stream nat) : Stream nat :=
  stream_zip mult σ τ .
Infix "⊙ " := stream_product (at level 40, left associativity).
```

resulting in a familiar set of stream operators analogous to those for lists and natural numbers.

As before, we also define two partial summation functions for streams,

```
CoFixpoint stream_partial_sums_acc (a : nat)
         (σ  : Stream nat) : Stream nat :=
  σ (0) + a ::: stream_partial_sums_acc (σ (0) + a) σ′.
```

and

```
Definition stream_partial_sums (σ  : Stream nat) : Stream nat :=
  stream_partial_sums_acc 0 σ .
```

which are dual to the existing list versions.

While we use a lot of properties of the different list and stream operators throughout the proofs of this dissertation, we first introduce these properties when they become relevant.

Now that we have introduced a stream calculus, which act as the dual of our list calculus, we finish the chapter by defining Moessner's sieve working on streams.

4.3 Moessner's sieve working on streams

In order to define Moessner's sieve working on streams, we take a step back and go over the description of the traditional Moessner's sieve working on a sequence of values. Moessner's sieve is a procedure which takes an initial sequence of values and a natural number, n, and generates a corresponding result sequence of successive powers, by dropping every nth element of the initial sequence and partially summing the remaining elements into a new sequence. This step of dropping and partially summing is repeated $n-1$

times on the intermediate result sequences, where n is decreased by 1 for each iteration; the sieve stops when n reaches 1. We refer to n as the rank of the sieve. For example, the application of Moessner's sieve of rank 5 on the sequence of 1s yields a result sequence of powers of 4,

$$\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & \ldots \\
1 & 2 & 3 & \mathbf{4} & & 5 & 6 & 7 & \mathbf{8} & & 9 & 10 & 11 & \mathbf{12} & & \ldots \\
1 & 3 & \mathbf{6} & & & 11 & 17 & \mathbf{24} & & & 33 & 43 & \mathbf{54} & & & \ldots \\
1 & \mathbf{4} & & & & 15 & \mathbf{32} & & & & 65 & \mathbf{108} & & & & \ldots \\
\mathbf{1} & & & & & \mathbf{16} & & & & & \mathbf{81} & & & & & \ldots
\end{array} \tag{4.1}$$

where we have marked the elements which are dropped in the sieve with boldface, and arranged the intermediate sequences to be aligned with the initial sequence.

If we are to translate the above description into Coq, we start by representing the sequence of values as a `Stream` of `nat`s, and use the procedure `stream_partial_sums` for partially summing the elements not dropped in a step of the sieve. This still leaves the dropping part of the sieve step to be formalized, as such we define the stream operator `drop`,

```
CoFixpoint drop (i k : nat) (σ : Stream nat) : Stream nat :=
  match i with
    | 0 ⇒ (σ′)(0) ::: drop (k - 2) k σ″
    | S i' ⇒ σ (0) ::: drop i' k σ′
  end.
```

which partitions a `Stream` into blocks of size `k` and drops the `i`th element, indexed from 0, of every block. By combining `drop` and `stream_partial_-sums`, we can define the `sieve_step` procedure,

```
Definition sieve_step (i k : nat) (σ : Stream nat) :=
  stream_partial_sums (drop i k σ ).
```

which performs one step of dropping and partially summing as described above. Now, by repeating the application of `sieve_step` n times, we can define the `sieve` procedure,

```
Fixpoint sieve (i k n : nat) (σ : Stream nat) : Stream nat :=
  match n with
    | 0 ⇒ sieve_step i k σ
    | S n' ⇒ sieve_step i k (sieve (S i) (S k) n' σ )
  end.
```

which performs Moessner's sieve when given a pair of initial values for the drop indices, `i` and `k`, along with the rank, `n`, and a seed stream, σ. We note that in the definition of `sieve` the initial values of `i` and `k` denotes the size and drop index of the last application of `sieve_step`, as we increment these for every recursive call. To illustrate this, we can translate the above example of applying Moessner's sieve of rank 5 on a sequence of 1s into,

```
Str_prefix 3 (sieve 1 2 3 #1) = [1; 16; 81].
```

where `i = 1` and `k = 2`, meaning that we partition the second-to-last stream, returned from the recursive call to `sieve`, into blocks of size 2 where we drop the second element. This corresponds to the last two sequences of Figure 4.1,

$$\begin{array}{llllllll} 1 & \mathbf{4} & 15 & \mathbf{32} & 65 & \mathbf{108} & & \ldots \\ \mathbf{1} & & \mathbf{16} & & \mathbf{81} & & & \ldots \end{array}$$

Lastly, we note that the rank, `n`, passed to `sieve` is `3` instead of `5`, which is the cause of two things,

1. We apply `sieve_step` even in the base case of `sieve` where `n = 0`, and
2. we do not drop the values of the final result stream,

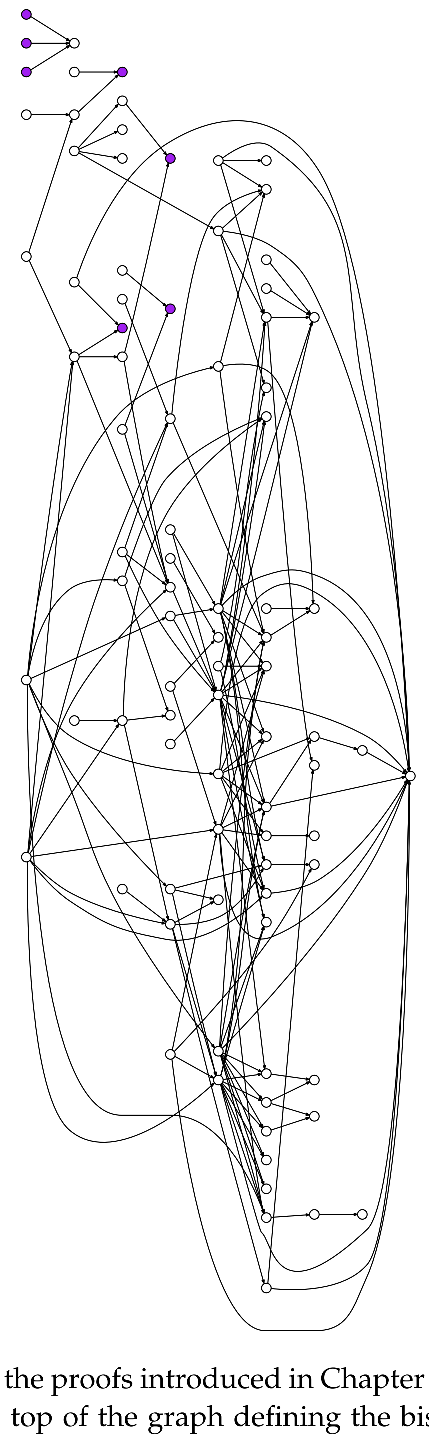
hence the difference of 2 between the rank value passed to `sieve` and the one applied in Figure 4.1.

Having defined the traditional version of Moessner's sieve working on streams, we are now ready to explore its dual in the next chapter, where we further analyze the operational description of Moessner's sieve.

4.4 Summary

In this chapter, we have established a list calculus and a stream calculus that constitute the foundation on which we build many of our later proofs, as the underlying mechanics of Moessner's sieve and its dual reduces to operations on lists and streams. Furthermore, we have introduced a procedure for generating the traditional version of Moessner's sieve working on streams.

The constructed list calculus consists of a `list` type along with a range of selectors, constructors, and operators, which allows us to examine individual elements of a list, create new lists from scratch or create new lists out of existing ones. Dually, the constructed stream calculus consists of a `Stream` type having the same types of selectors, constructors, and operators, but also its own notion of equality, called bisimilarity, as traditional Leibniz equality is too strict for proving equality between (most) coinductive types. Lastly, the version of Moessner's sieve working on streams is defined as a recursive procedure which repeatedly applies a composite stream operator consisting of a drop and partial summation operation.



Dependency graph of the proofs introduced in Chapter 4. Note the small cluster of theorems at the top of the graph defining the bisimilarity equivalence, and the density at the center of the graph reflecting the interdependence of our basic set of selectors, constructors and operators.

Chapter 5

A dual to Moessner's sieve

The poet doesn't invent.
He listens.

JEAN COCTEAU

The goal of this chapter is to introduce a dual to Moessner's sieve that simplifies the initial configuration of Moessner's sieve, by starting from two seed tuples instead of a stream, and creates a sequence of Moessner triangles, each constructed column by column, instead of a stream of successive powers, constructed row by row.

The chapter is structured as follows. In Section 5.1, we motivate the redefinition of Moessner's sieve as a procedure for generating a sequence of so-called Moessner triangles, instead of a stream of successive powers. As a result, we introduce two triangle creation procedures, which construct individual Moessner triangles either row by row or column by column, and prove equivalence between the two procedures. In Section 5.2, we first show how our triangle creation procedures give rise to a new and simpler initial configuration of Moessner's sieve, which we then use as inspiration for the final formalization of the dual of Moessner's sieve.

5.1 From streams to triangles

In this section, we first motivate the idea of looking at Moessner's sieve as a procedure for generating a sequence of Moessner triangles, as opposed to a stream of successive powers, and then formalize the core operation of the sieve that creates the individual Moessner triangles.

5.1.1 Generating a sequence of triangles with Moessner's sieve

In order to motivate the idea of redefining Moessner's sieve as a procedure for generating a sequence of Moessner triangles, we start by looking at an example application of Moessner's sieve working on streams.

Given a stream of 1s, let us apply Moessner's sieve of rank 5 on it,

$$\begin{array}{ccccccccccccccccc}
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & \ldots \\
1 & 2 & 3 & \mathbf{4} & & 5 & 6 & 7 & \mathbf{8} & & 9 & 10 & 11 & \mathbf{12} & & \ldots \\
1 & 3 & \mathbf{6} & & & 11 & 17 & \mathbf{24} & & & 33 & 43 & \mathbf{54} & & & \ldots \\
1 & \mathbf{4} & & & & 15 & \mathbf{32} & & & & 65 & \mathbf{108} & & & & \ldots \\
\mathbf{1} & & & & & \mathbf{16} & & & & & \mathbf{81} & & & & & \ldots
\end{array} \tag{5.1}$$

which yields the stream of successive powers of 4 (the rank minus one). As seen from this description, we traditionally view Moessner's sieve as a procedure which takes a seed stream consisting of an arithmetic progression, as first shown by Long [22], and returns a result stream of successive powers, by repeatedly dropping and partially summing the elements of the seed stream. Now, instead of focusing solely on the elements of the result stream, we want to view Moessner's sieve as generating a sequence of triangles, each of which we call a 'Moessner triangle'. The Moessner triangles appear as a result of preserving the alignment of the entries of the intermediate result streams in Moessner's sieve, while performing the repeated dropping of elements, as seen in Figure 5.1. Hence, we can pick the first triangle created in the above example,

$$\begin{array}{lllll}
[1 & 1 & 1 & 1 & 1] \\
[1 & 2 & 3 & 4] & \\
[1 & 3 & 6] & & \\
[1 & 4] & & & \\
[1] & & & &
\end{array} \tag{5.2}$$

and describe it as a set of tuples, marked by $[\ldots]$, which translates to the following notation in Coq,

```
Notation tuple := (list nat).
Notation triangle := (list tuple).
```

that we use throughout the dissertation when referring to Moessner triangles in the context of Moessner's sieve. Lastly, we say that a Moessner triangle's rank is equal to its depth minus one - or one less than the drop index of Moessner's sieve - and therefore the Moessner triangles of Formula 5.1 all have rank 4.

Thus, we have now defined what we mean by a Moessner triangle in the context of Moessner's sieve and defined a notation for the `triangle` and `tuple` types. In the next sections, we define procedures for creating individual tuples and combining them into triangles.

5.1.2 Make tuple

Before defining our `make_tuple` procedure, we first return to the triangles in Figure 5.1 and observe that we can view each of them as being constructed from two seed tuples: a horizontal seed tuple corresponding to a slice of the seed stream, and a vertical seed tuple corresponding to the dropped elements, marked with boldface, of the previous triangle. For example, the triangle shown in Figure 5.2 can be seen as the result of adding a horizontal seed tuple of 1s and a vertical seed tuple of 0s, since no elements have been dropped before the first triangle,

$$\begin{tikzcd}
 & {[1} \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & {1]} \\
\overset{\sqcap}{0} \arrow[r] & {[1} \arrow[r] \arrow[d] & 2 \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {4]} \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {6]} \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & {4]} \\
0 \arrow[r] & {[1]} \\
\underset{\sqcup}{0}
\end{tikzcd} \qquad (5.3)$$

From Figure 5.3, we notice that the rows and columns of the triangle are created in the exact same fashion – as the partial sums of the previous row/-column together with an accumulator. Specifically, the rth horizontal tuple (row) of the result triangle is created by partially summing all but the last entry of the $(r-1)$th horizontal tuple, while using the rth value of the vertical seed tuple as the accumulator value. For example, we can obtain the second horizontal tuple, $[1,3,6]$, of the result triangle, by partially summing the first horizontal tuple, $[1,2,3,4]$,

$$\begin{tikzcd}
 & 1 \arrow[d] & 2 \arrow[d] & 3 \arrow[d] & 4 \\
0 \arrow[r] & 1 \arrow[r] & 3 \arrow[r] & 6
\end{tikzcd}$$

where we ignore the last element, 4, and use the second entry in the vertical tuple, 0, as the accumulator for the partial summation. The same approach can be used for obtaining the cth vertical tuple (column) by partially summing the $(c-1)$th vertical tuple. This is also the reason why we have added an extra 0 in the initial vertical seed tuple - to ensure symmetry with respect to this procedure. By reducing Moessner's sieve to this core operation, which works for both rows and columns, we can translate our description above into the following Coq `Fixpoint`,

```
Fixpoint make_tuple (xs : tuple) (a : nat) : tuple :=
  match xs with
  | [] ⇒ []
  | [x] ⇒ []
  | x :: (_ :: _) as xs' ⇒
    let a' := x + a
    in a' :: make_tuple xs' a'
  end.
```

which takes a `tuple`, `xs`, and a natural number, `a`, as the accumulator, and returns a new `tuple` as already described.

As discussed above, the main operation of `make_tuple` is partial summation which leads us to state equivalence relations between `make_tuple` and our existing partial summation functions, `stream_partial_sums_acc` and `list_partial_sums_acc`,

```
Theorem equivalence_of_make_tuple_and_stream_partial_sums_acc :
  ∀ (l' a : nat) (σ  : Stream nat),
    make_tuple (Str_prefix (S l') σ ) a =
    Str_prefix l' (stream_partial_sums_acc a σ ).
```

and

```
Theorem equivalence_of_make_tuple_and_list_partial_sums_acc :
  ∀ (xs : tuple) (a : nat),
    make_tuple xs a =
    removelast (list_partial_sums_acc a xs).
```

The proof of `equivalence_of_make_tuple_and_stream_partial_sums_acc` is done by induction on `l'`, which is the length of the prefix of the stream being partially summed. Likewise, the proof of `equivalence_of_make_tuple_and_list_partial_sums_acc` is done by structural induction on the tuple `xs`.

Having defined a procedure for creating tuples, corresponding to individual rows or columns in a Moessner triangle, we move on to define procedures for creating a whole triangle as a list of tuples.

5.1.3 Create triangle

As already mentioned in the previous section, we can construct individual rows or columns of a Moessner triangle using the same procedure, `make_tuple`, which means that we can create a triangle by either repeatedly applying `make_tuple` on the horizontal seed tuple while using the vertical seed tuple as the list of accumulator values, or vice versa,

$$\begin{tikzcd}
 & {[1} \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1] \\
\overset{\sqcap}{0} \arrow[r] & {[1} \arrow[r] \arrow[d] & 2 \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & 4] & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & 6] & & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 4] & & & \\
0 \arrow[r] & {[1]} & & & & \\
\underset{\sqcup}{0} & & & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & {[1} \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1] \\
\overset{\sqcap}{0} \arrow[r] & \overset{\sqcap}{1} \arrow[r] \arrow[d] & \overset{\sqcap}{2} \arrow[r] \arrow[d] & \overset{\sqcap}{3} \arrow[r] \arrow[d] & \overset{\sqcap}{4} & \\
0 \arrow[r] & 1 \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & \underset{\sqcup}{6} & & \\
0 \arrow[r] & 1 \arrow[r] \arrow[d] & \underset{\sqcup}{4} & & & \\
0 \arrow[r] & \underset{\sqcup}{1} & & & & \\
\underset{\sqcup}{0} & & & & &
\end{tikzcd}
\tag{5.4}$$

As a result of this observation, we define two triangle creation procedures that each take two `tuples`, `xs` and `ys`, corresponding to the horizontal seed tuple and vertical seed tuple, respectively, and repeatedly applies make_tuple on these. For the first procedure, create_triangle_horizontally, we repeatedly create a new tuple, based on the elements of xs, while using the values of ys as accumulators, and cons the result onto the result tuple. In this way, xs holds the intermediate result of each recursive call, while the head of ys is removed for each recursive call. The algorithm terminates when there is one element or less left in ys and the result of the procedure is a list of horizontal tuples representing a triangle. Translating the above description into Coq, we obtain the following Fixpoint,

```
Fixpoint create_triangle_horizontally (xs ys : tuple) : triangle :=
  match ys with
    | [] ⇒ []
    | [y] ⇒ []
    | y :: (_ :: _) as ys' ⇒
      let xs' := make_tuple xs y
      in xs' :: (create_triangle_horizontally xs' ys')
  end.
```

which creates a Moessner triangle in a row by row fashion. For the second procedure, create_triangle_vertically, we simply switch the roles of the xs and ys described above, and obtain the dual procedure,

```
Fixpoint create_triangle_vertically (xs ys : tuple) : triangle :=
  match xs with
    | [] ⇒ []
    | [x] ⇒ []
    | x :: (_ :: _) as xs' ⇒
      let ys' := make_tuple ys x
      in ys' :: (create_triangle_vertically xs' ys')
  end.
```

creating the same triangle represented as a list of vertical tuples. The duality of create_triangle_horizontally and create_triangle_vertically is now evident as the definitions of the two procedures are completely identical except that the xs and ys have switched roles.

Thus, we have now defined the inner working of the dual of Moessner's sieve, specifically the procedure create_triangle_vertically which works column by column, when creating a Moessner triangle, while the traditional version of Moessner's sieve works row by row and is based on streams. Since we have only observed that the two procedures create the same triangles, we now move on to prove that this is indeed true.

5.1.4 Equivalence of the two triangle creation procedures

While we have argued that the two triangle creation procedures create the same Moessner triangles, when given the same input, we have not yet proved this proposition to be true.

Before stating our equivalence proof, we first have to make a clear distinction between the visual representation of the triangles in this chapter and the representation used in our Coq scripts. As seen in Figure 5.4, we write the result tuples of the second triangle in a vertical fashion implying a matrix-like indexing of the triangles. However, for the representation in Coq, we simply return a list of tuples for both triangle creation procedures, which results in the procedure `create_triangle_vertically` actually creating the transposed triangle of `create_triangle_horizontally` with respect to indexing,

$$\begin{tikzcd}
 & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 & 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 2 \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {4]} & 1 \arrow[r] & {[1} \arrow[r] \arrow[d] & 1 \arrow[r] \arrow[d] & 1 \arrow[r] \arrow[d] & {1]} & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {6]} & & 1 \arrow[r] & {[2} \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {4]} & & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & {4]} & & & 1 \arrow[r] & {[3} \arrow[r] \arrow[d] & {6]} & & & \\
0 \arrow[r] & {[1]} & & & & 1 \arrow[r] & {[4]} & & & & \\
0 & & & & & 1 & & & & &
\end{tikzcd} \tag{5.5}$$

This distinction leads to the following equivalence relation between `create_triangle_horizontally` and `create_triangle_vertically`,

```
Theorem equivalence_of_vertical_and_horizontal_triangle_swap :
  ∀ (xs ys : tuple),
    (create_triangle_horizontally xs ys) =
    (create_triangle_vertically ys xs).
```

expressing the symmetric property that swapping the input of one procedure yields the result of the other, which can be seen in Figure 5.5. The proof of this theorem is done by structural induction on the second input tuple, `ys`, and case analysis on both tuples, `xs` and `ys`.

Alternatively, we can also define an equivalence relation in terms of the indices of the entries of the two triangles,

```
Theorem equivalence_of_vertical_and_horizontal_triangle_indices :
  ∀ (i j : nat) (xs ys : tuple),
    (length xs) = (length ys) →
    (nth i (nth j (create_triangle_horizontally xs ys) []) 0) =
    (nth j (nth i (create_triangle_vertically xs ys) []) 0).
```

where the entry `(j,i)` of the triangle created by `create_triangle_horizontally` is equal to the entry `(i,j)` of the triangle created by `create_triangle_vertically`, when given the same input, `xs` and `ys`. As a result, the theorem captures the fact that the created Moessner triangles are each others transposed. The proof of the theorem is done by induction on the row and column indices, `i` and `j`, along with case analysis on the structure of the two tuples, `xs` and `ys`. Thus, we have now proved that the two triangle creation procedures do indeed create the same Moessner triangle when given the same input, which proves the correctness of the first part of our dual of Moessner's sieve.

With the above equivalence proofs in hand, our next step is to formalize the dual of Moessner's sieve using the dual triangle creation procedure, `create_triangle_vertically`, introduced in this chapter.

5.2 The dual of Moessner's sieve

In this section, we formalize the dual of Moessner's sieve as a procedure for creating a list of Moessner triangles, using `create_triangle_vertically`, which starts from a minimal initial configuration. Hence, we first make the case for simplifying the initial configuration of Moessner's sieve and then define the procedures which combined yields the dual of Moessner's sieve.

5.2.1 Simplifying the initial configuration

Before proceeding to state the final dual of Moessner's sieve, we first investigate whether our new approach affords a simpler initial configuration of Moessner's sieve. Hence, we again turn our attention to the Moessner triangles in Figure 5.5, and notice that the seed tuple containing 1s, in both triangles, is only a part of the input and not a part of the output, which we ideally would like in order to properly mimic the traditional version of Moessner's sieve. Fortunately, we can solve this issue by adopting an idea posed by Danvy et al. [7], where we observe that the stream of 1s can be created by partially summing a stream of a 1 followed by 0s. Furthermore, this generalization of Moessner's sieve has the effect of making Moessner's sieve capable of computing streams of the 0th power, as opposed to streams of squares, which was the base case of Moessner's original theorem.

Now, we can apply this generalization to the case of the procedure `create_triangle_horizontally`,

$$\begin{tikzcd}
 & 1 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 1 \arrow[r] \arrow[d] & 1 \arrow[r] \arrow[d] & 1 \arrow[r] \arrow[d] & {1]} & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 2 \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {4]} & & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & 3 \arrow[r] \arrow[d] & {6]} & & & \\
0 \arrow[r] & {[1} \arrow[r] \arrow[d] & {4]} & & & & \\
0 \arrow[r] & {[1]} & & & & & \\
0 & & & & & &
\end{tikzcd}$$

where we divide the stream of a 1 followed by 0s into equally sized horizontal seed tuples, one for each triangle, and add an extra 0 to the vertical seed tuples.

If we use this simplified configuration for the two initial Moessner triangles of Figure 5.1,

$$
\begin{array}{ccccccccccccccc}
 & 1 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & & \\
0 & 1 & 1 & 1 & 1 & 1 & & 1 & 1 & 1 & 1 & 1 & 1 & \\
0 & 1 & 2 & 3 & 4 & & & 4 & 5 & 6 & 7 & 8 & & \\
0 & 1 & 3 & 6 & & & & 6 & 11 & 17 & 24 & & & \\
0 & 1 & 4 & & & & & 4 & 15 & 32 & & & & \\
0 & 1 & & & & & & 1 & 16 & & & & & \\
0 & & & & & & & 0 & & & & & &
\end{array}
\tag{5.6}
$$

we discover a consistent property where the seed tuples are always located outside of the result triangles, while the result triangles contain exactly the values we want to capture with the sieve. As such, we define the rank of a seed tuple to be equal to its length minus two - or the rank of the generated Moessner triangle - meaning that the seed tuples in Formula 5.6 all have rank 4.

However, we do notice a small inconsistency in the two triangles in Figure 5.6, since the initial horizontal seed tuple consists of a 1 followed by 0s while all subsequent horizontal seed tuples consist of plain 0s. Fortunately, we know from Long [22] and Hinze [14] that the first triangle created by Moessner's sieve is always Pascal's triangle, which allows us to swap the two seed tuples for the initial triangle, as Pascal's triangle is symmetric:

$$
\begin{array}{ccccccccccccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & & \\
1 & 1 & 1 & 1 & 1 & 1 & & 1 & 1 & 1 & 1 & 1 & 1 & \\
0 & 1 & 2 & 3 & 4 & & & 4 & 5 & 6 & 7 & 8 & & \\
0 & 1 & 3 & 6 & & & & 6 & 11 & 17 & 24 & & & \\
0 & 1 & 4 & & & & & 4 & 15 & 32 & & & & \\
0 & 1 & & & & & & 1 & 16 & & & & & \\
0 & & & & & & & 0 & & & & & &
\end{array}
\tag{5.7}
$$

Thus, we obtain an initial configuration where the horizontal seed tuples are always 0s, for all created Moessner triangles, and the whole sieve is created from a single seed value located at the top of the first vertical seed tuple.

Having reduced the initial configuration of Moessner's sieve to this extremely simple set of seed tuples, we are ready to define the last procedures needed to define our dual of Moessner's sieve.

5.2.2 Hypotenuse of triangles

As seen in Figure 5.7, the values of the hypotenuse of the first Moessner triangle, $[1,4,6,4,1]$, are used as the vertical seed tuple for the next Moessner triangle. Thus, we need a procedure which takes a `triangle` and returns its hypotenuse as a `list`. This is implemented in a straightforward fashion by going through each `tuple`, `t`, of a `triangle`, `ts`, and aggregating the last values of each `tuple` into a new `tuple`, which is then returned,

```
Fixpoint hypotenuse (ts : triangle) : tuple :=
  match ts with
    | [] ⇒ []
    | t :: ts' ⇒ (last t 0) :: (hypotenuse ts')
  end.
```

For example, if we feed the triangle,

$$\begin{array}{ccccc}
1 & 1 & 1 & 1 & 1 \\
5 & 6 & 7 & 8 & \\
11 & 17 & 24 & & \\
15 & 32 & & & \\
16 & & & &
\end{array}$$

to `hypotenuse`, we get the tuple, $[16,32,24,8,1]$, when reading it column by column.

Since we are mainly interested in defining the dual of Moessner's sieve, which creates a list of Moessner triangles, each constructed column by column, all we have left to do is compose `create_triangle_vertically` and `hypotenuse` into a procedure which creates a list of `triangles`.

5.2.3 Create triangles

By combining `create_triangle_vertically` and `hypotenuse` we can define a final procedure, `create_triangles_vertically`, which given two seed tuples, `xs` and `ys`, and a length argument, `n`, returns a list of `n` Moessner triangles. The procedure works by applying `create_triangle_vertically` on the two input tuples, `xs` and `ys`, which creates the initial Moessner triangle whose hypotenuse is then used as the `ys` seed tuple of the next triangle while the `xs` remain unchanged. For each triangle created, we decrement the value of `n` and terminate the procedure when `n = 0`. This description brings us to the following definition,

```
Fixpoint create_triangles_vertically (n : nat) (xs ys : tuple)
  : list triangle :=
  match n with
    | 0 ⇒ [create_triangle_vertically xs ys]
    | S n' ⇒
      let ts := create_triangle_vertically xs ys
      in ts :: (create_triangles_vertically n' xs
                 (rev (cons 0 (hypotenuse ts))))
  end.
```

which is exactly the dual of Moessner's sieve we wanted, since it creates a list of Moessner triangles by constructing one triangle at a time in a column by column fashion. Visualizing the sieve of Figure 5.1 using our new dual, yields the following three Moessner triangles,

	0	0	0	0	0	0		0	0	0	0	0	0		0	0	0	0	0	0
1	1	1	1	1	1		1	1	1	1	1	1		1	1	1	1	1	1	
0	1	2	3	4			4	5	6	7	8			8	9	10	11	12		
0	1	3	6				6	11	17	24				24	33	43	54			
0	1	4					4	15	32					32	65	108				
0	1						1	16						16	81					
0							0							0						

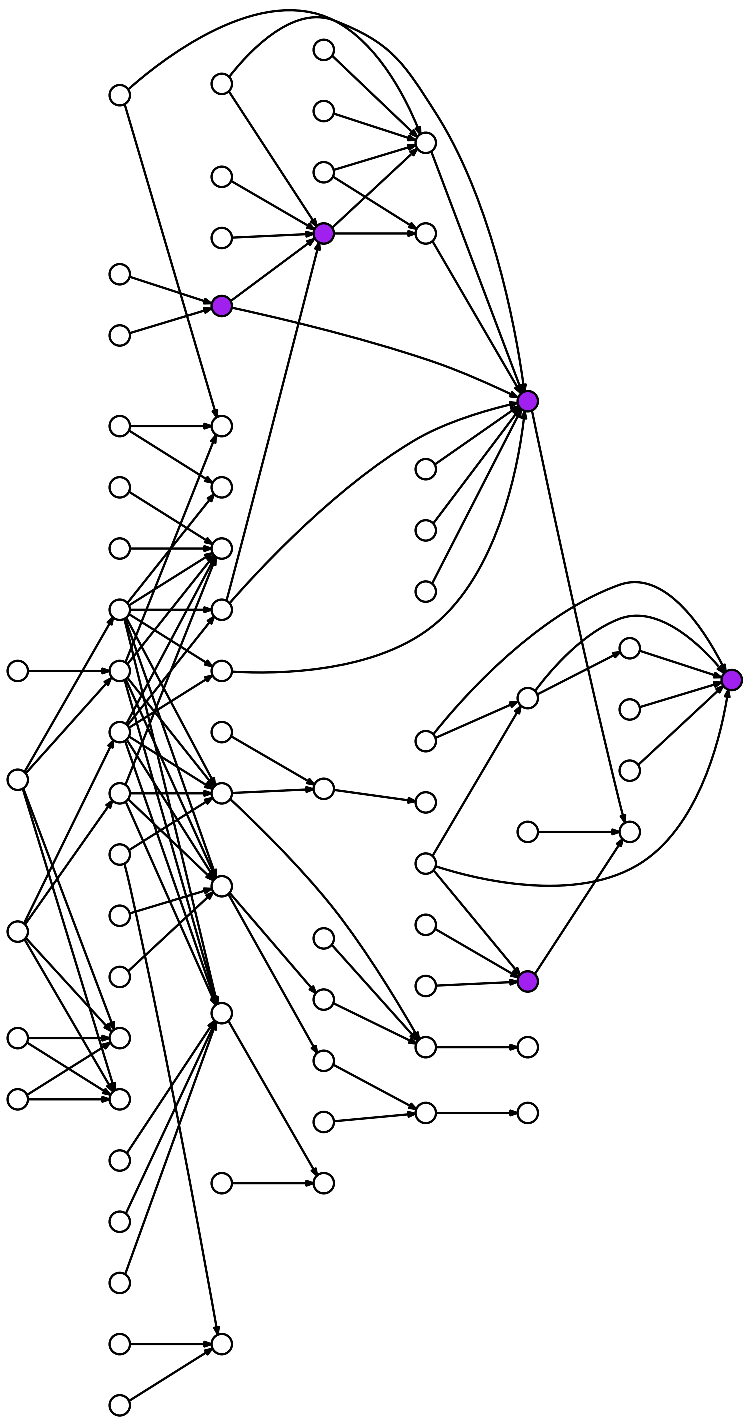
where we have explicitly added the seed tuples of each triangle.

Thus, we have now defined a dual to Moessner's sieve that creates a list of Moessner triangles, instead of a stream of successive powers, where each triangle is created column by column, instead of row by row, and which has an initial configuration consisting of two seed tuples having just one non-zero value, 1, located at the top of the vertical seed tuple, from which the whole sieve is subsequently constructed. In the remaining chapters of this dissertation, we abbreviate "the dual of Moessner's sieve" as simply "the dual sieve" on occasion.

5.3 Summary

In this chapter, we have introduced a dual to Moessner's sieve, which simplifies the initial configuration of Moessner's sieve, by starting from two seed tuples, and creates a sequence of Moessner triangles, each constructed column by column, instead of a stream of successive powers, constructed row by row.

The dual of Moessner's sieve was obtained by first observing that the traditional Moessner's sieve implicitly constructs triangles, called Moessner triangles, when we preserve the alignment of the elements of the intermediate result streams while repeatedly dropping elements in the streams. Combining this observation with the fact that each row and column in a Moessner triangle can be created using the same procedure, `make_tuple`, led to the definition of two symmetric triangle creation procedures, `create_triangle_horizontally` and `create_triangle_vertically`, each taking two tuples, one corresponding to a slice of a seed stream and one corresponding to the hypotenuse of the previous triangle, if any. By further combining the tuple-based approach with the observation that Moessner's sieve can be initialized from a stream of 1 followed by 0s, and the observation that the first triangle created by Moessner's sieve is always Pascal's triangle, resulted in a minimal initial configuration of Moessner's sieve starting from a single seed value, 1, while all other values of the respective seed tuples are 0. Lastly, by using the new initial configuration together with the procedure `create_triangle_vertically` paved the way for defining the dual procedure of Moessner's sieve, `create_triangles_vertically`, which creates a list of Moessner triangles instead of a stream of successive powers.



Dependency graph of the proofs introduced in Chapter 5. Note how the graph becomes more sparse as it progresses to the right, reflecting the concise flow of the proofs once the scaffolding is in place.

Chapter 6

Pascal's triangle and the binomial coefficient

Film is one of the three universal languages,
the other two: mathematics and music.

Frank Capra

The beautiful has its place in mathematics
for here are triumphs of the creative imagination,
beautiful theorems, proofs, and processes
whose perfection of form has made them classic.
He must be a 'practical' man
who can see no poetry in mathematics.

William F. White

The goal of this chapter is to introduce and formalize Pascal's triangle and the binomial coefficient function along with their rotated counterparts, as these describe the set of initial triangles generated by Moessner's sieve.

The chapter is structured as follows. In Section 6.1, we introduce and formalize Pascal's triangle and the binomial coefficient function, while we in the process also define the binomial theorem. Furthermore, we also prove an equivalence relation between Pascal's triangle and the binomial coefficient function. In Section 6.2, we motivate the introduction of the rotated Pascal's triangle and the rotated binomial coefficient function, both of which we formalize and prove an equivalence relation between. Furthermore, we also prove equivalence relations to the existing canonical versions of Pascal's triangle and the binomial coefficient function.

6.1 An introduction to Pascal's triangle and the binomial coefficient

In this section, we first give a basic description of Pascal's triangle followed by a definition of the binomial theorem, which we then use to motivate the introduction of the binomial coefficient function.

6.1.1 Pascal's triangle

Pascal's triangle is a triangular array named after the French mathematician Blaise Pascal, despite the fact that mathematicians in India, Greece, and China had studied the triangle several centuries prior to Pascal [10].

Traditionally, Pascal's triangle is indexed over its rows and their individual entries. The row index is denoted n and indexed from 0, referring to the top most row, while the entry index is denoted k and also indexed from 0, referring to the leftmost entry of a given row. We define $\binom{n}{k}$ to be the kth entry of the nth row of Pascal's triangle. Using a simple inductive approach we can construct Pascal's triangle as follows:

1. Let row 0 have a single entry, $\binom{0}{0}$, with the value 1.
2. For each entry k in the row n, $\binom{n}{k}$, calculate the value of the entry by adding the two entries just above it, i.e., the entries to its immediate left and right, $\binom{n-1}{k-1} + \binom{n-1}{k}$, and if one of the two entries does not exist, then replace its value with 0.

In order to strengthen our intuition, the figure below shows the first five rows of Pascal's triangle,

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

Here, we see that the entry $\binom{4}{2}$, whose value is 6, is calculated by adding the entries $\binom{3}{1}$ and $\binom{3}{2}$, both having the value 3, which again have been calculated by adding the values just above them, like so,

$$\begin{tikzcd}
1 \arrow[dr] & & 2 \arrow[dl] \arrow[dr] & & 1 \arrow[dl] \\
 & 3 \arrow[dr] & & 3 \arrow[dl] & \\
 & & 6 & &
\end{tikzcd}$$

Now, in order to translate Pascal's triangle into a Coq formalization, we first have to tweak the above inductive description a bit by making a few extra observations of the edge cases of Pascal's triangle. As such, we notice that the two outer legs only consists of 1s,

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

which gives us two alternative base cases,

$$\binom{n}{0} = 1, \tag{6.1}$$

and,

$$\binom{n}{n} = 1. \tag{6.2}$$

However, these do not capture the entries for which $n < k$, i.e., all values to the right of Pascal's triangle,

$$\begin{array}{cccccccccccc}
 & & & & 1 & & 0 & & 0 & & 0 & \dots \\
 & & & 1 & & 1 & & 0 & & 0 & & \dots \\
 & & 1 & & 2 & & 1 & & 0 & & 0 & \dots \\
 & 1 & & 3 & & 3 & & 1 & & 0 & & \dots \\
1 & & 4 & & 6 & & 4 & & 1 & & 0 & \dots
\end{array}$$

but, as seen in the above figure, we can simply state that,

$$\binom{n}{k} = 0, \text{ when } n < k. \tag{6.3}$$

If we combine the three base cases listed above with the following adjusted version of the inductive case described at the beginning of this section,

$$\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}, \tag{6.4}$$

which incidentally is called Pascal's rule, we are ready to formalize Pascal's triangle.

Now, instead of defining a type from which we can construct a whole triangle, we instead define an entry type, corresponding to an entry in any triangle,

```
Inductive Entry : Type :=
  | entry : nat → nat → nat → Entry.
```

which takes three natural numbers: its row index, its column index and its value. For example, the entry `(entry 4 2 6)` describes the entry $\binom{4}{2} = 6$ of Pascal's triangle. By focusing on a single entry of a triangle, we can define an inductive predicate that describes all entries of Pascal's triangle,

```
Inductive Pascal : Entry → Prop :=
| pascal_base_n_0 : ∀ (n : nat), (Pascal (entry n 0 1))
| pascal_base_n_n : ∀ (n : nat), 0 < n → (Pascal (entry n n 1))
| pascal_base_n_lt_k : ∀ (n k : nat), n < k → (Pascal (entry n k 0))
| pascal_inductive_S_n' : ∀ (n' k' v'' v' v : nat),
              v = v'' + v' →
              (Pascal (entry n' k' v'')) →
              (Pascal (entry n' (S k') v')) →
              (Pascal (entry (S n') (S k') v)).
```

where the first three cases,

```
| pascal_base_n_0 : ∀ (n : nat), (Pascal (entry n 0 1))
| pascal_base_n_n : ∀ (n : nat), 0 < n → (Pascal (entry n n 1))
| pascal_base_n_lt_k : ∀ (n k : nat), n < k → (Pascal (entry n k 0))
```

correspond to the three base cases in Formula 6.1-6.3, and the last case,

```
| pascal_inductive_S_n' : ∀ (n' k' v'' v' v : nat),
              v = v'' + v' →
              (Pascal (entry n' k' v'')) →
              (Pascal (entry n' (S k') v')) →
              (Pascal (entry (S n') (S k') v)).
```

corresponds to the adjusted inductive case in Formula 6.4. Thus, we have now described Pascal's triangle and translated it into an `Entry` type, which captures any entry of a triangle, along with an inductive predicate `Pascal`, which describes the set of entries in Pascal's triangle.

We now take a step back and define the binomial theorem, which we use to motivate the introduction of the binomial coefficient that is closely tied to Pascal's triangle, thus completing the circle.

6.1.2 The binomial theorem

Given a binomial $(x+y)^n$, where x and y are variables, and n is a natural number, we want to describe the expansion of $(x+y)^n$ as a sum of its terms, also called monomials.

If we examine the first four expansions of $(x+y)^n$, where $n = \{0,1,2,3\}$, we get the following equations,

$$\begin{aligned}
(x+y)^0 &= 1\\
(x+y)^1 &= x+y\\
(x+y)^2 &= x^2+xy+y^2\\
(x+y)^3 &= x^3+3x^2y+3xy^2+y^3,
\end{aligned}$$

from which we notice a pattern that can be made more distinct by explicitly writing every coefficient and exponent of every monomial in the expansions,

$$\begin{aligned}
(x+y)^0 &= 1x^0y^0 \\
(x+y)^1 &= 1x^1y^0 + 1x^0y^1 \\
(x+y)^2 &= 1x^2y^0 + 2x^1y^1 + 1x^0y^2 \\
(x+y)^3 &= 1x^3y^0 + 3x^2y^1 + 3x^1y^2 + 1x^0y^3.
\end{aligned} \tag{6.5}$$

Now it is clear that for a given binomial expansion the exponent of x is decremented for each monomial of the expansion, starting at n, as we traverse them from left to right. Conversely, the exponent of y, starting at 0, is incremented for each monomial of the expansion. If we let k denote the kth monomial of an expansion, we can generalize the observation just made to the sum,

$$\sum_{k=0}^{n} x^{n-k}y^k, \tag{6.6}$$

which generates the correct exponentiation of the monomials in the expansion. For example, if we let $n = 3$, we get,

$$\sum_{k=0}^{3} x^{3-k}y^k = x^3y^0 + x^2y^1 + x^1y^2 + x^0y^3,$$

which enumerates the monomials of the binomial expansion of $(x+y)^3$, in Formula 6.5, except for the coefficients of the monomials, which we still have to account for. Now, if we rearrange the monomials of Formula 6.5, such that they are vertically aligned around the same center,

$$\begin{array}{ccccccc}
 & & & 1x^0y^0 & & & \\
 & & 1x^1y^0 & & 1x^0y^1 & & \\
 & 1x^2y^0 & & 2x^1y^1 & & 1x^0y^2 & \\
1x^3y^0 & & 3x^2y^1 & & 3x^1y^2 & & 1x^0y^3
\end{array}$$

and remove everything but the coefficients,

$$\begin{array}{ccccccc}
 & & & 1 & & & \\
 & & 1 & & 1 & & \\
 & 1 & & 2 & & 1 & \\
1 & & 3 & & 3 & & 1
\end{array} \tag{6.7}$$

we obtain the first four rows of Pascal's triangle. Thus, the entries of Pascal's triangle enumerate the coefficients of the monomials in the binomial expansion of $(x+y)^n$. As such, we also refer to $\binom{n}{k}$ as a binomial coefficient, a concept we discuss further in the next section, since it gives us a way to calculate the coefficient of the kth monomial in the binomial expansion of $(x+y)^n$.

Returning to Formula 6.6, we can now combine it with the binomial coefficient $\binom{n}{k}$ and obtain the sum,

$$\sum_{k=0}^{n} \binom{n}{k} x^{n-k}y^k, \tag{6.8}$$

which calculates the binomial expansion of the expression $(x+y)^n$, yielding the following theorem.

Theorem 1 (Binomial theorem)**.** *Given two natural numbers x and y, and an exponent n, the expression $(x+y)^n$ can be expanded into the sum,*

$$(x+y)^n = \binom{n}{0}x^n y^0 + \binom{n}{1}x^{n-1}y^1 + \cdots + \binom{n}{n-1}x^1 y^{n-1} + \binom{n}{n}x^0 y^n,$$

where $\binom{n}{k}$ is the binomial coefficient. This expansion can also be written in summation notation as,

$$\begin{aligned}(x+y)^n &= \sum_{k=0}^{n}\binom{n}{k}x^{n-k}y^k \\ &= \sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k},\end{aligned} \tag{6.9}$$

where the last equivalence follows from the symmetry of the sequence of binomial coefficients and of x and y.

Later in this dissertation, we define functions in the Coq proof assistant to calculate the individual monomials of a binomial expansion and also prove a simplified version of the binomial theorem in which x is the only variable and y has been substituted by 1,

$$(1+x)^n = \sum_{k=0}^{n}\binom{n}{k}x^k. \tag{6.10}$$

Having defined the binomial theorem, we return to the binomial coefficient and show how to calculate and formalize it.

6.1.3 The binomial coefficient

As mentioned in the previous section, the binomial coefficient $\binom{n}{k}$ is equal to the coefficient of the kth monomial in the binomial expansion of $(x+y)^n$ and can be read from Pascal's triangle. This suggests that we can obtain a binomial coefficient function if we can reduce the predicate `Pascal` into something computable.

In order to come up with such a binomial coefficient function, we need to cover the base cases and inductive cases for the row and column indices, `n` and `k`. Hence, we first observe that the base case $\binom{n}{0} = 1$, covers the two cases where `(n = 0, k = 0)` and `(n = S n', k = 0)`, which leaves the cases `(n = 0, k = S k')` and `(n = S n', k = S k')`. For the case `(n = 0, k = S k')`, we know from the definition of `Pascal` that for all values $n < k$ the result is 0, and similarly we know that the case `(n = S n', k = S k')` is the sum of the two entries just above it `(n = n', k = S k')` and `(n = n', k = k')`. Combining all this we get the following Coq formalization,

```
Fixpoint binomial_coefficient (n k : nat) : nat :=
  match n, k with
    | n, 0 ⇒ 1
    | 0, S k' ⇒ 0
    | S n', S k' ⇒ binomial_coefficient n' (S k') +
                   binomial_coefficient n' k'
  end.
Notation "C( n , k )" := (binomial_coefficient n k).
```

where we use the alternative notation, `C(n, k)`, for the binomial coefficient function. To show that the properties of `Pascal` also holds for `binomial_-coefficient`, we prove Pascal's rule,

```
Theorem Pascal_s_rule' :
  ∀ (n' k' : nat),
    C(S n', S k') = C(n', S k') + C(n', k').
```

which follows from the definition of `binomial_coefficient`, and the property that all entries where $n < k$ are 0,

```
Lemma binomial_coefficient_n_lt_k_implies_0 :
  ∀ (n k : nat), n < k → C(n, k) = 0.
```

which we prove by induction on the row index, `n`, and case analysis on the column index, `k`. Finally, we prove that the value of `binomial_coefficient` is 1 when $n = n$,

```
Lemma binomial_coefficient_n_eq_k_implies_1 :
  ∀ (n : nat), C(n, n) = 1.
```

which is done by induction on the row index, `n`, and using the already proved `binomial_coefficient_n_lt_k_implies_0` property. Thus, we have now formalized the binomial coefficient function in Coq and proved basics properties about it.

Having defined both `Pascal` and the `binomial_coefficient` function, we move on to prove that they are indeed equivalent.

6.1.4 Equivalence of Pascal's triangle and the binomial coefficient function

When comparing the definitions of the `binomial_coefficient` function and the `Pascal` predicate, we observe that if we have an entry which is a Pascal entry, `Pascal (entry n k v)`, then its value, `v`, must be equal to the corresponding value of the binomial coefficient function, `v = C(n, k)`, which brings us to the following equivalence proof,

```
Theorem Pascal_iff_binomial_coefficient :
  ∀ (n k v : nat),
    Pascal (entry n k v) ↔ v = C(n, k).
```

which states that an `Entry` is in Pascal's triangle if and only if its value is equal to the corresponding value of the binomial coefficient function. We prove the implication from left to right,

```
Lemma Pascal_implies_binomial_coefficient :
  ∀ (n k v : nat),
    Pascal (entry n k v) → v = C(n, k).
```

by induction on the row index n and case analysis on the different clauses of the Pascal predicate, and likewise prove the implication from right to left,

```
Lemma binomial_coefficient_implies_Pascal :
  ∀ (n k v : nat),
    v = C(n, k) → Pascal (entry n k v).
```

by induction on the row index, n followed by case analysis on the entry index, k, where each subcase is matched to a corresponding clause of the Pascal predicate. Thus, we have now proved that there exists an equivalence relation between entries of Pascal's triangle and the values calculated by the binomial coefficient function.

Now that we have defined the canonical versions of Pascal's triangle and the binomial coefficient function, and formalized both in the Coq proof assistant, we are ready to introduce their rotated counterparts.

6.2 Rotating Pascal's triangle and the binomial coefficient

In this section, we introduce and formalize the rotated versions of Pascal's triangle and the binomial coefficient.

First, we compare Pascal's triangle with the triangles generated by Moessner's sieve, which we use to motivate the rotated Pascal's triangle. Due to the tight connection between Pascal's triangle and the binomial coefficient function, we also define the rotated binomial coefficient function and finally prove the equivalence between the rotated Pascal's triangle and the rotated binomial coefficient function.

6.2.1 Rotated Pascal's triangle

In order to motivate the introduction of the rotated Pascal's triangle, we start by examining the triangles generated by applying Moessner's sieve of rank 5 on a sequence of 1s,

1	1	1	1	**1**	1	1	1	1	**1**	1	1	1	1	**1**	...
1	2	3	**4**		5	6	7	**8**		9	10	11	**12**		...
1	3	**6**			11	17	**24**			33	43	**54**			...
1	**4**				15	**32**				65	**108**				...
1					**16**					**81**					...

from which we notice that the initial Moessner triangle is in fact Pascal's triangle that has been rotated, as seen in the figure below,

$$\begin{array}{lllll} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & \\ 1 & 3 & 6 & & \\ 1 & 4 & & & \\ 1 & & & & \end{array} \qquad \begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

This observation can be made for all ranks of Moessner's sieve, and has previously been made by Long [22] and Hinze [15], which encourages us to formalize this rotated version of Pascal's triangle. Now, if we index the triangle in terms of its rows and columns, denoted r and c, both indexed from 0, we can state similar properties capturing the characteristics of the rotated Pascal's triangle, as we did in the case of `Pascal`.

First, we notice that the base case of Pascal's triangle, where $\binom{n}{0} = 1$, now corresponds to the first column of the rotated Pascal's triangle,

$$\begin{array}{lllll} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & \\ 1 & 3 & 6 & & \\ 1 & 4 & & & \\ 1 & & & & \end{array}$$

Furthermore, the case where $\binom{n}{k} = 0$, when $n < k$, has now disappeared due to the rotation transformation making us unable to index outside of the triangle, and the case $\binom{n}{n} = 1$ now corresponds to the first row of the rotated Pascal's triangle,

$$\begin{array}{lllll} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & \\ 1 & 3 & 6 & & \\ 1 & 4 & & & \\ 1 & & & & \end{array}$$

Lastly, for the inductive case, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, where we previously added the two entries just above the entry we wanted to calculate, we now add its immediate western and northern neighbors,

$$\begin{tikzcd} & & 1 \arrow[d] \\ & 2 \arrow[r] \arrow[d] & 3 \arrow[d] \\ 1 \arrow[r] & 3 \arrow[r] & 6 \end{tikzcd}$$

Combining these observations, we can formalize a predicate which captures all entries in the rotated Pascal's triangle,

```
Inductive Rotated_Pascal : Entry → Prop :=
| rotated_pascal_base_r_0 :
   ∀ (r : nat), (Rotated_Pascal (entry r 0 1))
| rotated_pascal_base_0_c :
   ∀ (c : nat), 0 < c → (Rotated_Pascal (entry 0 c 1))
| rotated_pascal_induction_r_c :
   ∀ (r' c' v'' v' v : nat),
     v = v'' + v' →
     (Rotated_Pascal (entry (S r') c' v'')) →
     (Rotated_Pascal (entry r' (S c') v')) →
     (Rotated_Pascal (entry (S r') (S c') v)).
```

in a manner analogously to the definition of `Pascal`. With this predicate, we have now motivated the rotated Pascal's triangle and formalized it in Coq as a predicate describing all entries in the rotated triangle.

Just as we defined a binomial coefficient function as a dual to Pascal's triangle, we now move on to define a rotated binomial coefficient function as a dual to the rotated Pascal's triangle.

6.2.2 Rotated binomial coefficient

If we were to try and define the rotated binomial coefficient function in a straight-forward manner, where we translated the clauses of `Rotated_Pascal` into base- and inductive cases for the row and column indices, `r` and `c`, like so,

```
Fixpoint rotated_binomial_coefficient (r c : nat) : nat :=
  match r, c with
    | r, 0 ⇒ 1
    | 0, c ⇒ 1
    | S r', S c' ⇒ rotated_binomial_coefficient r' (S c') +
                   rotated_binomial_coefficient (S r') c'
  end.
```

we would not be allowed to by the Coq proof assistant, since it would not be able to guess the decreasing argument of our recursive call. The reason why it cannot guess the decreasing argument is because of the inductive case containing two recursive calls, in which the first call decreases the row index while the second call decreases the column index. So, instead we have to define the rotated binomial coefficient function in terms of the existing `binomial_coefficient` function.

In order to do so, we first introduce the notation `R(r, c)` to mean the rotated binomial coefficient function applied on the row index, `r`, and column index, `c`, which we then use to capture the relation to the existing `binomial_-coefficient` function, `C(n, k)`. First, we observe that the row index `r` maps directly to the row index `n`, since we have the following relation of the base cases `R(r, 0) = C(r, 0) = 1`. Furthermore, if we let `r = 0`, we notice that the column index `c` maps to both `n` and `k` as `R(0, c) = C(c, c)` as also observed in the previous section. Putting these relations together yields the following formalization of the rotated binomial coefficient function,

```
Definition rotated_binomial_coefficient (r c : nat) : nat :=
  C(c + r, c).
Notation "R( r , c )" := (rotated_binomial_coefficient r c).
```

Now that we have defined `rotated_binomial_coefficient`, we can prove some of its properties as already characterized in `Rotated_Pascal`. First, we show that Pascal's rule also holds in a rotated version,

```
Theorem rotated_Pascal_s_rule' :
  ∀ (r' c' : nat),
    R(S r', S c') = R(S r', c') + R(r', S c').
```

which follows by case analysis on the row index, `r`. Furthermore, we also prove that the two base cases of `Rotated_Pascal` hold for `rotated_binomial_coefficient`,

```
Lemma unfold_rotated_binomial_coefficient_base_case_r_0 :
  ∀ (r : nat),
    R(r, 0) = 1.

Lemma unfold_rotated_binomial_coefficient_base_case_0_c :
  ∀ (c : nat),
    0 < c → R(0, c) = 1.
```

Here, we prove the first property by case analysis on the row index, `r`, along with the existing `binomial_coefficient_n_eq_k_implies_1` property, while the second property is proved by case analysis on the column index, `c`.

Lastly, we prove that `rotated_binomial_coefficient` is symmetric,

```
Theorem rotated_binomial_coefficient_is_symmetric :
  ∀ (r c : nat), R(r, c) = R(c, r).
```

which we do by induction on the row and column indices, `r` and `c`, while using the already proved properties of `rotated_binomial_coefficient`. As a corollary, we prove that `binomial_coefficient` is also symmetric,

```
Theorem binomial_coefficient_is_symmetric :
  ∀ (n k : nat),
    k ≤ n →
    C(n, k) = C(n, n - k).
```

by rewriting with the following equivalence relation between `binomial_coefficient` and `rotated_binomial_coefficient`,

```
Corollary binomial_coefficient_eq_rotated_binomial_coefficient :
  ∀ (n k : nat),
    k ≤ n →
    C(n, k) = R(n - k, k).
```

which captures the same relation as found in the definition of `rotated_binomial_coefficient`.

Now that we have introduced both the rotated Pascal's triangle and the rotated binomial coefficient function, all we have left to do is prove an equivalence relation between these two in a manner similar to what we did for their canonical counterparts.

6.2.3 Equivalence of the rotated Pascal's triangle and the rotated binomial coefficient

If we once again compare the definitions of `rotated_binomial_coefficient` and `Rotated_Pascal`, we note that the value of an entry in the rotated Pascal's triangle, `Rotated_Pascal (entry r c v)`, must be equal to the value calculated by `rotated_binomial_coefficient`, `v = R(r, c)`, which yields an equivalence relation completely analogous to the one for `binomial_coefficient` and `Pascal`,

```
Theorem Rotated_Pascal_iff_rotated_binomial_coefficient :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) ↔ v = R(r, c).
```

Furthermore, the two implications proofs are also proved in similar ways to the ones for the canonical versions of Pascal's triangle and the binomial coefficient function. As such the proof of the implication from left to right,

```
Lemma Rotated_Pascal_implies_rotated_binomial_coefficient :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) → v = R(r, c).
```

follows by induction on the row index, `r`, and case analysis on the different clauses of `Rotated_Pascal` in the base case, and induction on the column index, `c`, in the inductive case. Likewise, the proof of the implication from right to left,

```
Lemma rotated_binomial_coefficient_implies_Rotated_Pascal :
  ∀ (r c v : nat),
    v = R(r, c) → Rotated_Pascal (entry r c v).
```

follows by induction on the row index, `r`, and case analysis on the column index, `c`, where each subcase is matched to the corresponding clause of the `Rotated_Pascal` predicate. This proves the equivalence relation of `Rotated_Pascal` and `rotated_binomial_coefficient`.

As in the case of `rotated_binomial_coefficient`, we prove that the rotated Pascal's triangle is symmetric,

```
Theorem Rotated_Pascal_is_symmetric :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) ↔
    Rotated_Pascal (entry c r v).
```

which we do by substituting occurrences of `Rotated_Pascal` with `rotated_binomial_coefficient` and using `rotated_binomial_coefficient_is_symmetric`. Likewise, we can prove that `Pascal` is symmetric,

```
Theorem Pascal_is_symmetric :
  ∀ (n k v : nat),
    k ≤ n →
    (Pascal (entry n k v) ↔
     Pascal (entry n (n - k) v)).
```

as we have also proved that `binomial_coefficient` is symmetric.

Lastly, now that we have proved equivalence between `Rotated_Pascal` and `rotated_binomial_coefficient`, and between `rotated_binomial_-coefficient` and `binomial_coefficient`, we can transitively obtain the proof that an `Entry` is a `Pascal` entry if and only if it is also a `Rotated_-Pascal` entry,

```
Theorem Rotated_Pascal_iff_Pascal :
  ∀ (n k v : nat),
    Rotated_Pascal (entry n k v) ↔
    Pascal (entry (n + k) k v).
```

which we have stated in such a way that we can ignore all values outside of the canonical Pascal's triangle. The proofs of the two implications,

```
Corollary Pascal_implies_Rotated_Pascal :
  ∀ (n k v : nat),
    Pascal (entry (n + k) k v) →
    Rotated_Pascal (entry n k v).
```

and

```
Corollary Rotated_Pascal_implies_Pascal :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) →
    Pascal (entry (r + c) c v).
```

follow as corollaries of the already proved equivalence relations mentioned in this and the previous section.

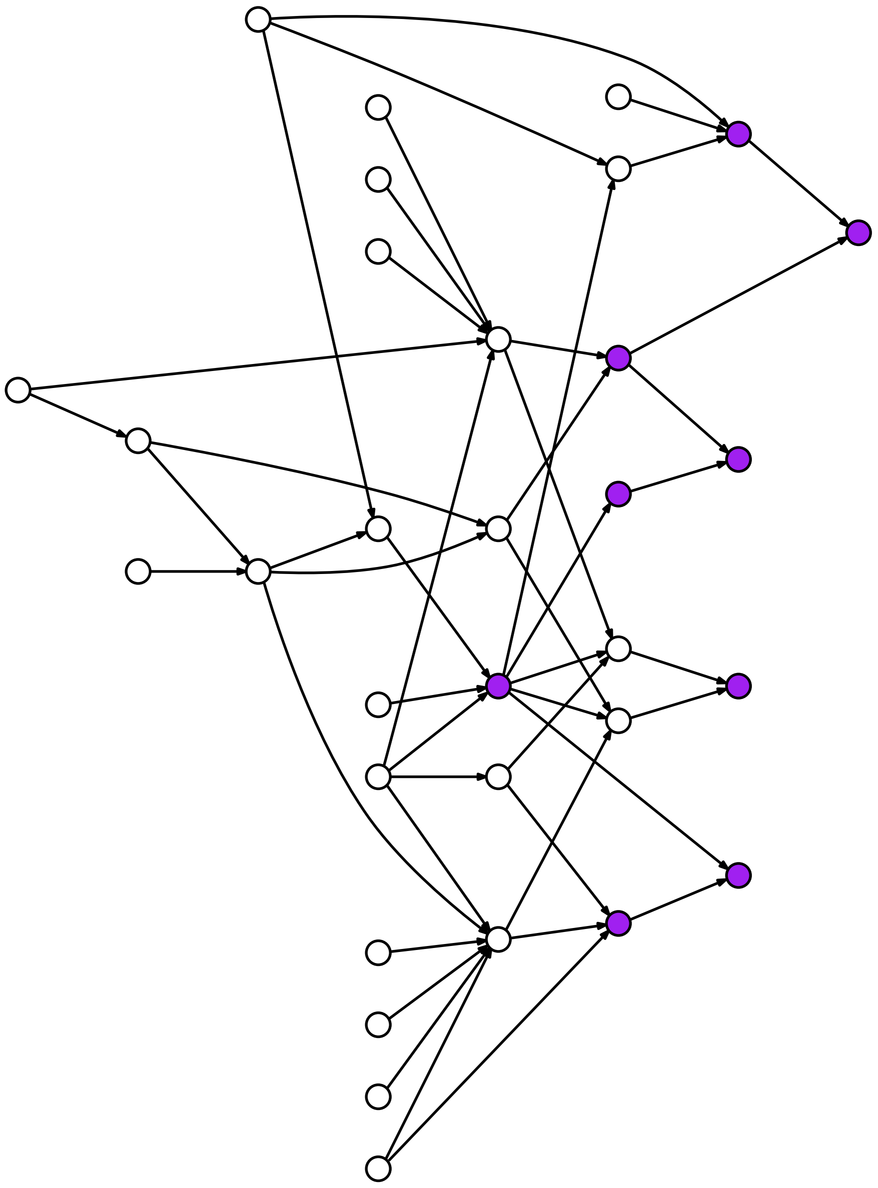
We have now reached the goal of this chapter, as we have formalized Pascal's triangle and the binomial coefficient function along with their rotated counterparts. Thus, we are now ready to combine what we have learned in this and the previous chapter in order to characterize the triangles generated by Moessner's sieve in the next chapter.

6.3 Summary

In this chapter, we have introduced and formalized Pascal's triangle and the binomial coefficient function along with their rotated counterparts, which describe the set of initial triangles generated by Moessner's sieve.

We obtained the above results by first describing the construction of Pascal's triangle in an inductive fashion, followed by translating it into a Coq formalization as the inductive type, `Entry`, and the inductive predicate, `Pascal`.

Afterwards, we introduced the binomial coefficient function, as a result of describing the binomial theorem, and formalized it as the function `binomial_-coefficient`, which we proved to have an equivalence relation with `Pascal`. Having formalized the canonical versions of Pascal's triangle and the binomial coefficient function, we motivated the introduction of their rotated counterparts by examining the triangles generated by Moessner's sieve, and noticing that the first triangle generated is always equal to Pascal's triangle. Consequently, we defined a similar predicate for the rotated Pascal's triangle, `Rotated_Pascal`, and the rotated binomial coefficient function, `rotated_-binomial_coefficient`, as in the canonical case, and proved an equivalence relation between the two, while also proving an equivalence relation between the canonical and rotated formalizations.



Dependency graph of the proofs introduced in Chapter 6. Note how the sparsity of the graph reflects the simplicity of the formalizations and the connections between them.

Chapter 7

A characteristic function of Moessner's sieve

It might be worth-while to point out
that the purpose of abstracting is not to be vague,
but to create a new semantic level
in which one can be absolutely precise.

Edsger W. Dijkstra, 1972 (EWD340)

The goal of this chapter is to introduce a characteristic function of Moessner's sieve, which computes the entries of a given Moessner triangle without needing to compute the prefix of the sieve. Furthermore, we present a correctness proof for the characteristic function, which shows that it computes the same Moessner triangles as created by the dual sieve. In the same vein, we also prove the correctness of the triangle creation procedure of the dual sieve with respect to the traditional sieve procedure that works on streams.

The chapter is structured as follows. In Section 7.1, we derive the operational description of a characteristic function of Moessner's sieve, which we then formalize in Section 7.2 as the two Coq procedures `moessner_entry` and `rotated_moessner_entry`. Following this, we prove a relation between the repeated application of `make_tuple` on a seed tuple of a given Moessner triangle, expressed as a binomial expansion, and the resulting columns of the created triangle, expressed in terms of the characteristic function `rotated_moessner_entry`, in Section 7.3. Building on the proofs of Section 7.3, we prove the correctness of our characteristic functions, `moessner_entry` and `rotated_moessner_entry`, in Section 7.4, through the construction of an auxiliary procedure, `repeat_make_tuple`, which bridges the proof gap between the triangle creation procedures, `create_triangle_horizontally` and `create_triangle_vertically`, and the characteristic functions, `moessner_entry` and `rotated_moessner_entry`. Lastly, with the help of the auxiliary procedure `repeat_make_tuple`, we also prove the correctness of the triangle creation procedures with respect to the traditional sieve procedure, `sieve`, thus verifying that our dual of Moessner's sieve is indeed valid.

7.1 Characterizing Moessner triangles

In order to come up with a characteristic function of the triangles generated by Moessner's sieve, we first have to uncover the patterns by which they are constructed. Hence, let us examine the first three Moessner triangles created by applying Moessner's sieve of rank 5 – yielding powers of 4 – on the stream of 1s,

$$\begin{array}{ccccccccccccccc}
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} \\
1 & 2 & 3 & \mathbf{4} & & 5 & 6 & 7 & \mathbf{8} & & 9 & 10 & 11 & \mathbf{12} & \\
1 & 3 & \mathbf{6} & & & 11 & 17 & \mathbf{24} & & & 33 & 43 & \mathbf{54} & & \\
1 & \mathbf{4} & & & & 15 & \mathbf{32} & & & & 65 & \mathbf{108} & & & \\
\mathbf{1} & & & & & \mathbf{16} & & & & & \mathbf{81} & & & &
\end{array} \tag{7.1}$$

and see if we can discover any properties that help us characterize the triangles. As previously pointed out in this dissertation and by Hinze [14], we can observe that the initial triangle generated by Moessner's sieve is always equal to the rotated Pascal's triangle, having a depth equal to the rank of the Moessner triangle plus one. Furthermore, we also notice that the subsequent Moessner triangles exhibit Pascal-like properties, i.e., Pascal's rule holds for all triangles, as every entry is the sum of its immediate western and northern neighbors, as previously illustrated in Figure 5.3 and originally noted by Long [22]. Knowing that the Moessner triangles behave in a Pascal-like fashion, hints at a possible binomial coefficient-like characteristic function, parameterized over the first row and column of a given Moessner triangle. If we again focus on Figure 7.1, it is trivial to see that the first row of every Moessner triangle is filled with 1s, while we need to discover a new property in order to characterize the first column of every triangle.

Returning to the initial Moessner triangle, we know from the equivalence between Pascal's triangle and the binomial coefficient, that the hypotenuse of the triangle will always enumerate the coefficients of the monomials of the binomial expansion of $(1+t)^r$, where r is equal to the rank of the triangle, and t is a variable. Using the initial Moessner triangle of Figure 7.1 as an example, we get the binomial expansion,

$$(1+t)^4 = 1 \cdot t^4 + 4 \cdot t^3 + 6 \cdot t^2 + 4 \cdot t^1 + 1 \cdot t^0,$$

where the values of the hypotenuse, $(1, 4, 6, 4, 1)$, do indeed enumerate the binomial coefficients of the expansion. Incidentally, the hypotenuse also enumerates the actual terms of the binomial expansion when $t = 1$,

$$\begin{aligned}(1+1)^4 &= 1 \cdot 1^4 + 4 \cdot 1^3 + 6 \cdot 1^2 + 4 \cdot 1^1 + 1 \cdot 1^0 \\ &= 1 + 4 + 6 + 4 + 1,\end{aligned}$$

which raises the question of what happens if we let t denote the triangle index, starting from $t = 1$. As it turns out, letting $t = 2$,

$$\begin{aligned}(1+2)^4 &= 1 \cdot 2^4 + 4 \cdot 2^3 + 6 \cdot 2^2 + 4 \cdot 2^1 + 1 \cdot 2^0 \\ &= 16 + 32 + 24 + 8 + 1\end{aligned}$$

results in the terms of the binomial expansion to be equal to the values found in the hypotenuse of the second Moessner triangle, $(16, 32, 24, 8, 1)$, in Figure 7.1. We can observe that this property holds for all triangles,

$$\begin{aligned}(1+3)^4 &= 1\cdot 3^4 + 4\cdot 3^3 + 6\cdot 3^2 + 4\cdot 3^1 + 1\cdot 3^0\\ &= 81 + 108 + 54 + 12 + 1,\end{aligned}$$

as seen here for $t = 3$, and was recently pointed out by Danvy et al. [7] as a characterization of the values dropped in the individual triangles of Moessner's sieve. Combining this observation with the fact that the entries of a Moessner triangle are created using Pascal's rule, leads us to the realization that the first column of the $(1+t)$th Moessner triangle enumerates the partial sums of the monomials of the binomial expansion $(1+t)^r$,

$$\begin{array}{lllll} 1 & 1 & 1 & 1 & 1\\ 1 & 2 & 3 & 4 & \\ 1 & 3 & 6 & & \\ 1 & 4 & & & \\ 1 & & & & \end{array} \Rightarrow \begin{array}{lllll} 1 & 1 & 1 & 1 & 1\\ 5 & 6 & 7 & 8 & \\ 11 & 17 & 24 & & \\ 15 & 32 & & & \\ 16 & & & & \end{array} \Rightarrow \begin{array}{lllll} 1 & 1 & 1 & 1 & 1\\ 9 & 10 & 11 & 12 & \\ 33 & 43 & 54 & & \\ 65 & 108 & & & \\ 81 & & & & \end{array} \tag{7.2}$$

as seen in Figure 7.2, where $(1, 4, 6, 4, 1)$ partially sums to $(1, 5, 11, 15, 16)$, and $(1, 8, 24, 32, 16)$ partially sums to $(1, 9, 33, 65, 81)$.

Having characterized how every Moessner triangle is constructed using Pascal's rule, where the first row of a triangle is a sequence of 1s and the first column is a partial sum parameterized over the binomial expansion, we are ready to formalize the characteristic function in Coq.

7.2 Defining a characteristic function

Synthesizing the observations made in the previous section, we now present two characteristic functions of Moessner's sieve. First, we define a characteristic function that is analogous to our existing `binomial_coefficient` function, `moessner_entry`, followed by a rotated version, `rotated_moessner_entry`, defined in terms of the first characteristic function, `moessner_entry`.

7.2.1 Moessner entry

In order to translate the informal description of the characteristic function made in Section 7.1 into a valid Coq function, we first define it as a function that is analogous to the existing `binomial_coefficient` function. As such, we rotate the first two Moessner triangles of Figure 7.1 into a Pascal-like configuration,

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
\qquad
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 5 & & 1 & & & \\
 & & 11 & & 6 & & 1 & & \\
 & 15 & & 17 & & 7 & & 1 & \\
16 & & 32 & & 24 & & 8 & & 1
\end{array}
\tag{7.3}
$$

where we use the same row-and-entry indexing scheme, `n` and `k`, as in the case of the `binomial_coefficient` function,

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

Just like the `binomial_coefficient` function, we have four combinations of `n` and `k` being either `0` or the successor of some `n'` or `k'`, where the only case that is different from the `binomial_coefficient` function is the one where `n = S n'` and `k = 0`, corresponding to the first column of a rotated Moessner triangle as discussed in the previous section. While we simply return `1` in the case of the `binomial_coefficient` function, we instead have to add the appropriate monomial of the last row of the previous triangle. For example, in Figure 7.3 the value `11` in the third row of the second triangle is obtained by adding `5`, located immediately above it, and the value `6`, located at the third entry of the last row of the previous triangle. This is the exact same behavior as we saw in Figure 7.2, but for the rotated Moessner triangles.

Combining the logic for the four cases of `n` and `k` yields the following binomial coefficients-like characteristic function of a Moessner triangle,

```
Fixpoint moessner_entry (r n k t : nat) : nat :=
  match n with
    | 0 ⇒ match k with
            | 0 ⇒ 1
            | S k' ⇒ 0
          end
    | S n' ⇒ match k with
               | 0 ⇒ monomial t r (S n') +
                     moessner_entry r n' 0 t
               | S k' ⇒ moessner_entry r n' (S k') t +
                        moessner_entry r n' k' t
             end
  end.
```

indexed using the row and column indices `n` and `k`, where `r` denotes the rank of the triangle and `t` the triangle index. The `monomial` function, used in the inductive case of `n`, is defined as,

```
Definition monomial (t r n : nat) : nat :=
  C(r, n) * (t ^ n).
```

and computes a single monomial of the binomial expansion $(1+t)^r$, when given a triangle index, `t`, a rank, `r`, and an index, `n`, of a monomial in the expansion.

Given `moessner_entry`'s strong ties to the `binomial_coefficient` function, it is no surprise that we can also prove that it exhibits some of the same properties. As observed, Pascal's rule holds true for all values of `moessner_-entry`,

```
Theorem moessner_entry_Pascal_s_rule :
  ∀ (n' r k' t : nat),
    moessner_entry r (S n') (S k') t =
    moessner_entry r n' (S k') t +
    moessner_entry r n' k' t.
```

and so do the following two properties,

```
Lemma moessner_entry_n_eq_k_implies_1 :
  ∀ (n r t : nat),
    moessner_entry r n n t = 1.
```

and

```
Lemma moessner_entry_n_lt_k_implies_0 :
  ∀ (r n k t : nat),
    n < k →
    moessner_entry r n k t = 0.
```

completely analogous to the lemmas proved in the case of the `binomial_-coefficient` function, for describing the entries of the right-most leg and entries outside of the triangle. Furthermore, we can now state and prove the repeatedly mentioned equivalence relation between the initial Moessner triangle and Pascal's triangle, by showing that the `binomial_coefficient` function and `moessner_entry` compute the same values when `t = 0`,

```
Theorem moessner_entry_eq_binomial_coefficient :
  ∀ (n k r : nat),
    moessner_entry r n k 0 = C(n, k).
```

The equivalence proof follows by induction on the row index, n, and case analysis on the entry index, k, thus proving the assumed connection between Moessner's sieve and Pascal's triangle.

Lastly, we define a new Stream which enumerates a row of a Pascal-like Moessner triangle,

```
CoFixpoint moessner_entries (r n k t : nat) : Stream nat :=
  (moessner_entry r n k t) :::
  (moessner_entries r n (S k) t).
```

using our newly created characteristic function, moessner_entry. The relation between moessner_entries and moessner_entry is captured by the following property,

```
Lemma Str_nth_moessner_entries :
  ∀ (i r n k t : nat),
    Str_nth i (moessner_entries r n k t) =
    moessner_entry r n (i + k) t.
```

which allows us to go from an element indexed by moessner_entries to a value computed by moessner_entry. The proof of the above property follows by induction on the element index, i, and rewriting according to the initial value and stream derivative of moessner_entries. We use moessner_entries later in this dissertation, when we prove Moessner's theorem.

Having defined a binomial coefficient-like characteristic function for the triangles generated by Moessner's sieve, we move on to define its rotated counterpart.

7.2.2 Rotated Moessner entry

Since the binomial_coefficient function and moessner_entry exhibit the same triangular structure, the relation between moessner_entry and its rotated counterpart, rotated_moessner_entry, is identical to the existing relation between binomial_coefficient and rotated_binomial_coefficient,

```
Definition rotated_binomial_coefficient (r c : nat) : nat :=
  C(c + r, c).
Notation "R( r , c )" := (rotated_binomial_coefficient r c).
```

Thus, we define the rotated version of moessner_entry like so,

```
Definition rotated_moessner_entry (n r c t : nat) : nat :=
  moessner_entry n (c + r) c t.
```

where n denotes the rank, r the row, c the column, and t the triangle. Furthermore, rotated_moessner_entry has similar properties to the ones discussed in the previous section, for moessner_entry,

```
Lemma rotated_moessner_entry_Pascal_s_rule :
  ∀ (n r' c' t : nat),
    rotated_moessner_entry n (S r') (S c') t =
    rotated_moessner_entry n r' (S c') t +
    rotated_moessner_entry n (S r') c' t.
```

along with

```
Lemma rotated_moessner_entry_r_eq_0_implies_1 :
  ∀ (c n t : nat),
    rotated_moessner_entry n 0 c t = 1.
```

and

```
Lemma rotated_moessner_entry_c_eq_0 :
  ∀ (r' n t : nat),
    rotated_moessner_entry n (S r') 0 t =
    monomial t n (S r') + rotated_moessner_entry n r' 0 t.
```

which we use extensively when reasoning about `rotated_moessner_entry` in many of our later proofs. Lastly, there also exists a similar equivalence relation between the rotated Pascal's triangle and `rotated_moessner_entry`, when `t = 0`,

```
Corollary rotated_moessner_entry_eq_rotated_binomial_coefficient :
  ∀ (n r c : nat),
    rotated_moessner_entry n r c 0 = R(r, c).
```

which follows as a corollary of `moessner_entry_eq_binomial_coefficient`.

As in the case of `moessner_entry`, we also create a `Stream` which enumerates a specific column of a Moessner triangle,

```
CoFixpoint rotated_moessner_entries (n r c t : nat) : Stream nat :=
  (rotated_moessner_entry n r c t) :::
  (rotated_moessner_entries n (S r) c t).
```

using the characteristic function `rotated_moessner_entry`. It too has the following indexing property,

```
Lemma Str_nth_rotated_moessner_entries :
  ∀ (i n r c t : nat),
    Str_nth i (rotated_moessner_entries n r c t) =
    rotated_moessner_entry n (r + i) c t.
```

along with a version of Pascal's rule,

```
Lemma rotated_moessner_entries_Pascal_s_rule :
  ∀ (n r' c' t : nat),
    rotated_moessner_entries n (S r') (S c') t ∼
    (rotated_moessner_entries n (S r') c' t) ⊕
    (rotated_moessner_entries n r' (S c') t).
```

working on streams created by `rotated_moessner_entries`, which we prove by coinduction and rewriting according to Pascal's rule for `rotated_-moessner_entry`.

Now that we have defined our two characteristic functions of Moessner's sieve, `moessner_entry` and `rotated_moessner_entry`, we turn our attention towards proving the correctness of these characteristic functions. However, in order to do so, we first have to construct the needed scaffolding on which to build the correctness proofs. Thus, our next step is to introduce procedures that describe the seed tuples of the Moessner triangles generated by Moessner's sieve, in terms of the monomials of a binomial expansion. We then use these procedures to characterize the columns of the triangles generated by applying our triangle creation procedures on the seed tuples.

7.3 Moessner entry and monomials

In this section, we prove a relation between the `c`th column of a Moessner triangle and the monomials of its vertical seed tuple from which it is created.

In order to do so, we first define procedures for explicitly enumerating the monomials of a binomial expansion, `monomials`, and their partial sums, `monomials_sum`, which we then use to state and prove an equivalence relation between the first column of a Moessner triangle, enumerated by `rotated_-moessner_entries`, and its seed tuple, expressed in terms of the defined monomial procedures. Having shown how to go from a vertical seed tuple to the first column of a Moessner triangle, we then prove how to obtain the `(S c')`th column, enumerated by `rotated_moessner_entries`, when given the `c'`th column, thus providing us with a base case and inductive case for a later correctness proof of `rotated_moessner_entry`.

7.3.1 Monomials and monomials sum

As mentioned in the previous sections, we have observed that the values of the hypotenuse of a triangle generated by Moessner's sieve,

$$\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & \mathbf{1} & \dots \\
1 & 2 & 3 & \mathbf{4} & & 5 & 6 & 7 & \mathbf{8} & & 9 & 10 & 11 & \mathbf{12} & & \dots \\
1 & 3 & \mathbf{6} & & & 11 & 17 & \mathbf{24} & & & 33 & 43 & \mathbf{54} & & & \dots \\
1 & \mathbf{4} & & & & 15 & \mathbf{32} & & & & 65 & \mathbf{108} & & & & \dots \\
\mathbf{1} & & & & & \mathbf{16} & & & & & \mathbf{81} & & & & & \dots
\end{array}$$

acts as the seed tuple for the subsequent triangle, and enumerates the individual monomials of the binomial expansion of $(1+t)^r$, where r is the rank of the Moessner triangle and t is the triangle index. Furthermore, we also know that the first column of the $(1+t)$th triangle is enumerated by the partial sums of its seed tuple, corresponding to the partial sums of the mentioned monomials. Hence, if we are to reason about the seed tuples of a given Moessner triangle, we have to introduce procedures which compute the corresponding monomials. As such, we define the procedure `monomials`,

```
CoFixpoint monomials (t r n : nat) : Stream nat :=
  (monomial t r n) ::: (monomials t r (S n)).
```

which enumerates the monomials of a binomial expansion when given a triangle index, t, a rank, r, and a start index, n, of the expansion. Similarly, we define the procedure monomials_sum,

```
CoFixpoint monomials_sum (t r n a : nat) : Stream nat :=
  let a' := (monomial t r n) + a in
  a' ::: (monomials_sum t r (S n) a').
```

which takes the same arguments as monomials along with an accumulator, a, and enumerates the partial sums of the same binomial expansion, from a given start index, n.

Lastly, we want to prove that the stated relation of monomials_sum being the partial sums of monomials is indeed correct,

```
Lemma stream_partial_sums_acc_monomials_bisim_monomials_sum :
  ∀ (t r n a : nat),
    stream_partial_sums_acc a (monomials t r n) ~
    monomials_sum t r n a.
```

which is done by coinduction using just the definitions of the two streams' initial values and stream derivatives for rewriting.

Now that we have defined procedures for enumerating the monomials of the binomial expansion, and their partial sums, we move on to prove an equivalence relation between the seed tuple of a Moessner triangle, expressed in terms of monomials, and its initial column, expressed in terms of rotated_-moessner_entries.

7.3.2 The first column of a Moessner triangle

As a result of the last proof in the previous section, stream_partial_sums_-acc_monomials_bisim_monomials_sum, and the following equivalence relation between make_tuple and stream_partial_sums_acc,

```
Theorem equivalence_of_make_tuple_and_stream_partial_sums_acc :
  ∀ (l' a : nat) (σ : Stream nat),
    make_tuple (Str_prefix (S l') σ) a =
    Str_prefix l' (stream_partial_sums_acc a σ).
```

which we have previously proved, we obtain the corollary,

```
Corollary make_tuple_monomials_eq_monomials_sum :
  ∀ (l t r n a : nat),
    make_tuple (Str_prefix (S l) (monomials t r n)) a =
    Str_prefix l (monomials_sum t r n a).
```

which states that applying make_tuple on a prefix of monomials yields a prefix of monomials_sum. Furthermore, we can also prove that the first column of a Moessner triangle, generated by our characteristic function, rotated_-moessner_entries, is bisimilar to monomials_sum,

```
Corollary rotated_moessner_entries_bisim_monomials_sum :
  ∀ (n t : nat),
    monomials_sum t n 0 0 ∼
    rotated_moessner_entries n 0 0 t.
```

for the same rank, n, and triangle index, t, which follows from the theorem,

```
Theorem Str_nth_monomials_sum_eq_rotated_moessner_entry :
  ∀ (r t n a : nat),
    Str_nth r (monomials_sum t n 0 a) =
    (rotated_moessner_entry n r 0 t) + a.
```

combined with Str_nth_rotated_moessner_entries, proved in Section 7.2.2, and the fact that element-wise equality implies bisimilarity. In order to prove Str_nth_monomials_sum_eq_rotated_moessner_entry, we do induction on the row index, r, and use the following Pascal-like property of monomials_-sum,

```
Lemma shift_start_index_monomials_sum :
  ∀ (i' r n a t : nat),
    (monomial t r n) + (Str_nth i' (monomials_sum t r (S n) a)) =
    (monomial t r (n + (S i'))) + (Str_nth i' (monomials_sum t r n a)).
```

which allows us to shift the start index of monomials_sum to monomial in certain cases. The proof of shift_start_index_monomials_sum follows by induction on the element index, i, combined with the following stream derivative property of monomials_sum,

```
Lemma Str_nth_monomials_sum_stream_derivative :
  ∀ (i n t l a : nat),
    Str_nth i (monomials_sum t l n a)′ =
    Str_nth i ((monomials_sum t l (S n) a) ⊕ #(monomial t l n)).
```

which is also proved by induction on the element index i.

Combining make_tuple_monomials_eq_monomials_sum and rotated_-moessner_entries_bisim_monomials_sum gives us the following corollary,

```
Corollary make_tuple_monomials_eq_rotated_moessner_entries :
  ∀ (l' n' t : nat),
    make_tuple (Str_prefix (S l') (monomials t n' 0)) 0 =
    Str_prefix l' (rotated_moessner_entries n' 0 0 t).
```

which clearly states that applying make_tuple on a prefix of monomials, which is also the content of our vertical seed tuples, yields the first column of a corresponding Moessner triangle, expressed in terms of our characteristic function rotated_moessner_entries. Thus, we have proved the first relation between the inner workings of our triangle creation procedure create_triangle_vertically, specifically make_tuple, and our characteristic function rotated_moessner_entries, and monomials, which describe the hypotenuse – or seed tuple – of a given Moessner triangle.

As we have now proved the base case of our characterization, which states that the first column of any Moessner triangle is equal to the partial sums of the monomials enumerated by the hypotenuse of the previous triangle, which in turn is also equal to the first column enumerated by rotated_moessner_-entries, we move on to tackle the inductive step which relates the c'th and (S c')th column of a Moessner triangle, in terms of rotated_moessner_-entries.

7.3.3 The subsequent columns of a Moessner triangle

Given that the subsequent columns of a Moessner triangle are created by partial summation using make_tuple, as seen in the definition of create_-triangle_vertically, we can simply state that partially summing the c'th column enumerated by rotated_moessner_entries yields the (S c')th column,

```
Corollary partial_sums_rotated_moessner_entries_bisim_next_column :
  ∀ (n c' t : nat),
    (stream_partial_sums (rotated_moessner_entries n 0 c' t)) ~
    (rotated_moessner_entries n 0 (S c') t).
```

since we have already proved the base case, c = 0, in terms of rotated_-moessner_entries. As in the previous section, we can prove the bisimilarity by showing that the two streams are element-wise equal,

```
Theorem Str_nth_partial_sums_rotated_moessner_entries :
  ∀ (i n c' t : nat),
    Str_nth i (stream_partial_sums
                (rotated_moessner_entries n 0 c' t)) =
    Str_nth i (rotated_moessner_entries n 0 (S c') t).
```

which is done by induction on the element index, i, and rewriting according to the already proved properties of rotated_moessner_entry and the lemma,

```
Lemma Str_nth_rotated_moessner_entries_over_r :
  ∀ (i n r' c' t : nat),
    Str_nth i (rotated_moessner_entries n (S r') (S c') t) =
    Str_nth i (stream_partial_sums_acc
                (rotated_moessner_entry n r' (S c') t)
                (rotated_moessner_entries n (S r') c' t)).
```

which is itself proved by induction on the element index, i.

Lastly, we can prove that applying make_tuple on the cth column of rotated_moessner_entries yields the (S c)th column,

```
Corollary make_tuple_rotated_moessner_entries :
  ∀ (l' n' c' t : nat),
    make_tuple (Str_prefix
                 (S l') (rotated_moessner_entries n' 0 c' t)) 0 =
    Str_prefix l' (rotated_moessner_entries n' 0 (S c') t).
```

due to the equivalence proof of make_tuple and stream_partial_sums, which, combined with the proof that the first column of any Moessner triangle is enumerated by rotated_moessner_entries,

```
Corollary make_tuple_monomials_eq_rotated_moessner_entries :
  ∀ (l' n' t : nat),
    make_tuple (Str_prefix (S l') (monomials t n' 0)) 0 =
    Str_prefix l' (rotated_moessner_entries n' 0 0 t).
```

gives us both the base case and inductive case of an implicit correctness proof of `rotated_moessner_entry`, as the characteristic function of Moessner's sieve.

Next, we combine the proofs of this section with the introduction of a new construct, `repeat_make_tuple`, which bridges the gap between `create_-triangle_vertically` and `rotated_moessner_entry`, in order to state a proper correctness proof of `rotated_moessner_entry`.

7.4 Correctness of Moessner entry

Building on the work of the previous sections, we now prove the correctness of our characteristic function, `rotated_moessner_entry`, by first extracting a new procedure, `repeat_make_tuple`, from `create_triangle_vertically` which simplifies its core and bridges the gap in the equivalence proof of `create_triangle_vertically` and `rotated_moessner_entry`. Furthermore, we also prove an equivalence relation between Moessner's sieve working on streams, `sieve`, and our triangle creation procedures, `create_triangle_-horizontally` and `create_triangle_vertically`, thus completing the chain from the traditional version of Moessner's sieve working on streams to our characteristic function.

7.4.1 Simplifying the mechanics of create triangle vertically

Instead of trying to directly prove the correctness of `rotated_moessner_-entry` with respect to `create_triangle_vertically`, we first investigate whether our current definitions afford the introduction of simpler procedures on which to build the correctness proof.

Returning to the definition of `create_triangle_vertically`,

```
Fixpoint create_triangle_vertically (xs ys : tuple) : triangle :=
  match xs with
    | [] ⇒ []
    | [x] ⇒ []
    | x :: (_ :: _) as xs' ⇒
      let ys' := make_tuple ys x
      in ys' :: (create_triangle_vertically xs' ys')
  end.
```

we note that reasoning about its behavior requires us to prove two base cases along with an inductive case, which involves two tuples and the construction of a `triangle`. However, if we could instead lift the core operation out of the procedure and into a simpler one, we could reduce the complexity of our proofs.

We observe that by setting the xs of create_triangle_vertically to always be 0s, as in the case of the dual sieve, we can simplify the mechanics of create_triangle_vertically to be the repeated application of make_tuple on a vertical seed tuple, ys, with the accumulator value 0. This brings us to the following procedure,

```
Fixpoint repeat_make_tuple (ys : tuple) (a n : nat) : tuple :=
  match n with
    | 0 ⇒ ys
    | S n' ⇒ repeat_make_tuple (make_tuple ys a) a n'
  end.
```

which only takes a single seed tuple, ys, an accumulator, a, and the number of applications of make_tuple, n, while presenting a much cleaner structure than create_triangle_vertically, as it only has one base case and one inductive case, and returns a tuple rather than a whole triangle.

Given that we have already proved how we can go from a seed tuple, expressed in terms of monomials, to the initial column of a Moessner triangle, expressed in terms of rotated_moessner_entries, and likewise go from the c'th column to the (S c')th column of a Moessner triangle, again expressed in terms of rotated_moessner_entries, we seem to have a valid candidate for stating and proving a general correctness proof of rotated_moessner_entry.

7.4.2 More columns of Moessner triangles

Having reduced the essence of create_triangle_vertically to the procedure repeat_make_tuple, we are now in a position to formalize the natural extension of make_tuple_rotated_moessner_entries,

```
Theorem repeat_make_tuple_rotated_moessner_entries :
  ∀ (c l n t : nat),
    repeat_make_tuple
      (Str_prefix (c + l) (rotated_moessner_entries n 0 0 t)) 0 c =
    Str_prefix l (rotated_moessner_entries n 0 c t).
```

which states that if we start from the first column generated by rotated_moessner_entries, of length (c + l), and apply make_tuple c times, we get the cth column of rotated_moessner_entries of length l, as we remove one element of the tuple per application of make_tuple.

The proof follows by induction on the column index, c, and the following helper lemma,

```
Lemma shift_make_tuple_in_repeat_make_tuple :
  ∀ (ys : tuple) (n a : nat),
    make_tuple (repeat_make_tuple ys a n) a =
    repeat_make_tuple (make_tuple ys a) a n.
```

which formalizes an associative property of make_tuple over repeat_make_tuple, and is proved by induction on the number of application of make_-

tuple, n. Lastly, due to the relation between monomials an rotated_-moessner_entries, we obtain the following corollary,

```
Corollary repeat_make_tuple_monomials_eq_moessner_entries :
  ∀ (c l n t : nat),
   repeat_make_tuple
     (Str_prefix (S c + l) (monomials t n 0)) 0 (S c) =
   Str_prefix l (rotated_moessner_entries n 0 c t).
```

from our proof of repeat_make_tuple_rotated_moessner_entries, which states that applying make_tuple (S c) times on a prefix of monomials, representing our set of potential seed tuples, yields a corresponding prefix of the cth column enumerated by rotated_moessner_entries.

With this strong relation between repeat_make_tuple and rotated_-moessner_entries in our hands, all we need to do to bridge the gap between create_triangle_vertically and rotated_moessner_entry, in order to prove the correctness of rotated_moessner_entry, is to prove that repeat_make_tuple is equivalent to create_triangle_vertically, when xs is a tuple of 0s.

7.4.3 Correctness proofs of repeat-make-tuple and rotated-moessner-entry

While we have proved that applying repeat_make_tuple on monomials yields a result in terms of rotated_moessner_entries, we still need to prove that repeat_make_tuple does indeed have an equivalence relation to create_-triangle_vertically in order to finish our correctness proofs of rotated_-moessner_entry.

We formulate the equivalence relation between repeat_make_tuple and create_triangle_vertically by stating that applying make_tuple on a tuple, ys, (S j) times yields the same result as the jth column of the triangle created by applying create_triangle_vertically on ys as the vertical seed tuple and a tuple of 0s as the horizontal seed tuple,

```
Theorem correctness_of_repeat_make_tuple :
  ∀ (j : nat) (ys : tuple),
   (repeat_make_tuple ys 0 (S j)) =
   (nth j (create_triangle_vertically
            (tuple_constant (length ys) 0)
            ys)
        []).
```

The proof is done by induction on the column index, j, followed by case analysis on the tuple, ys. Lastly, as in the case of repeat_make_tuple_rotated_-moessner_entries, we also need a shift-like property of make_tuple with respect to create_triangle_vertically,

```
Lemma shift_make_tuple_create_triangle_vertically :
  ∀ (j' : nat) (ys : tuple),
   make_tuple
     (nth j'
          (create_triangle_vertically
             (tuple_constant (length ys) 0) ys) []) 0 =
   nth (S j')
       (create_triangle_vertically
          (tuple_constant (length ys) 0) ys) [].
```

which is also proved by induction on the column index j, thus finishing the proof correctness_of_repeat_make_tuple.

Having proved the correctness of repeat_make_tuple, we now have an equivalence proof between rotated_moessner_entry and repeat_make_-tuple, and an equivalence proof between repeat_make_tuple and create_-triangle_vertically, thus all we need to prove the correctness of rotated_-moessner_entry is to compose the two proofs into one. As such, our final proof states that the ith entry of the jth column created by create_-triangle_vertically, when applied on a vertical seed tuple characterized by monomials, can be computed by rotated_moessner_entry when letting the row index be equal to i and the column index equal to j,

```
Theorem correctness_of_rotated_moessner_entry :
  ∀ (i j r t : nat),
    j ≤ S r →
    S i ≤ S r - j →
    (nth i
        (nth j
            (create_triangle_vertically
               (tuple_constant (S (S r)) 0)
               (Str_prefix (S (S r)) (monomials t r 0)))
            [])
        0) =
    rotated_moessner_entry r i j t.
```

The theorem follows by rewriting according to correctness_of_repeat_-make_tuple, proved above, and Str_nth_rotated_moessner_entries, proved in Section 7.2.2, along with a modified version of repeat_make_-tuple_monomials_eq_moessner_entries,

```
Lemma repeat_make_tuple_monomials_eq_moessner_entries_general :
  ∀ (k j n t : nat),
    j ≤ k →
    Str_prefix (k - j) (rotated_moessner_entries n 0 j t) =
    repeat_make_tuple (Str_prefix (S k) (monomials t n 0)) 0 (S j).
```

Proving the generalization of repeat_make_tuple_monomials_eq_-moessner_entries is done by induction on the indices k and j, while rewriting with the existing proofs of repeat_make_tuple_monomials_-eq_moessner_entries, shift_make_tuple_in_repeat_make_tuple, and make_tuple_rotated_moessner_entries. This proofs the correctness of our characteristic function.

Having proved correctness_of_rotated_moessner_entry, we are now certain that our characteristic function is indeed correct within the bounds of a given Moessner triangle created by create_triangle_vertically. However, we still have not proved the correctness of create_triangle_vertically with respect to sieve.

7.4.4 Correctness proof of create-triangle-vertically

In Chapter 4 we introduced Moessner's sieve working on streams, represented by sieve_step and sieve,

```
Definition sieve_step (i k : nat) (σ : Stream nat) :=
 stream_partial_sums (drop i k σ).
```

and

```
Fixpoint sieve (i k n : nat) (σ : Stream nat) : Stream nat :=
 match n with
   | 0 ⇒ sieve_step i k σ
   | S n' ⇒ sieve_step i k (sieve (S i) (S k) n' σ)
 end.
```

which we now prove to be equivalent to make_tuple and repeat_make_tuple, respectively, thus transitively proving the correctness of our triangle creation procedures, create_triangle_horizontally and create_triangle_-vertically.

In order to state an equivalence relation between make_tuple and sieve_-step, we first observe that sieve_step behaves like stream_partial_sums, for all stream prefixes having length less than the drop index i. Furthermore, we have already proved an equivalence relation between make_tuple and stream_partial_sums_acc,

```
Theorem equivalence_of_make_tuple_and_stream_partial_sums_acc :
  ∀ (l' a : nat) (σ : Stream nat),
    make_tuple (Str_prefix (S l') σ) a =
    Str_prefix l' (stream_partial_sums_acc a σ).
```

allowing us to formulate the following theorem,

```
Theorem equivalence_of_make_tuple_and_sieve_step_gen :
  ∀ (l i k a : nat) (σ : Stream nat),
    l ≤ i →
    make_tuple (Str_prefix (S l) σ) a =
    Str_prefix l ((sieve_step i k σ) ⊕ #a).
```

which states that applying make_tuple on the prefix of a stream, σ, with an accumulator, a, is equivalent to the prefix of the sum of sieve_step applied to σ and the constant stream #a, when the length of the prefix is less than or equal to the drop index, `l ≤ i`. The proof follows from the equivalence between `make_tuple` and `stream_partial_sums_acc` and the following helper lemma,

```
Lemma Str_prefix_drop_gen :
  ∀ (l i k : nat) (σ  : Stream nat),
    l ≤ i →
    Str_prefix l (drop i k σ ) =
    Str_prefix l σ .
```

which captures the observation made between `Str_prefix` and `drop`. As a corollary, we get the slightly simpler equivalence relation,

```
Corollary equivalence_of_make_tuple_and_sieve_step :
  ∀ (l : nat) (σ  : Stream nat),
    make_tuple (Str_prefix (S l) σ ) 0 =
    Str_prefix l ((sieve_step l (S l) σ )).
```

between `make_tuple` and `sieve_step`.

Now that we have captured the relation between `sieve_step` and `make_tuple`, we are in position to state an equivalence relation between `sieve` and `repeat_make_tuple`,

```
Theorem equivalence_of_repeat_make_tuple_and_sieve :
  ∀ (i n : nat) (σ  : Stream nat),
    n ≤ i →
    repeat_make_tuple (Str_prefix (S i) σ ) 0 (S n) =
    Str_prefix (i - n) (sieve i (S i) n σ ).
```

We prove the equivalence between `repeat_make_tuple` and `sieve` by case analysis on the drop index, `i`, followed by induction on the number of `make_tuple` applications, `n`. Besides `equivalence_of_make_tuple_and_sieve_step`, the proof also relies on a helper lemma similar to the one between `Str_prefix` and `drop`,

```
Lemma Str_prefix_sieve :
  ∀ (n l i : nat) (σ  : Stream nat),
    l ≤ i →
    Str_prefix l (sieve (S i) (S (S i)) n σ ) =
    Str_prefix l (sieve (S (S i)) (S (S (S i))) n σ ).
```

which essentially states that the prefix of two different applications of `sieve` on the same stream, σ, are equivalent as long as the drop index `i` is greater than or equal to the prefix length `l`. The proof follows by induction on the number of applications of `sieve_step`, `n`.

Lastly, due to the correctness of `repeat_make_tuple` and the equivalence of `create_triangle_vertically` and `create_triangle_horizontally`, we can prove that `sieve` is also equivalent to `create_triangle_horizontally`,

```
Corollary equivalence_of_sieve_and_create_triangle_horizontally :
  ∀ (n i : nat) (σ : Stream nat),
    n ≤ i →
    Str_prefix (i - n) (sieve i (S i) n σ) =
    nth n (create_triangle_horizontally
             (Str_prefix (S i) σ)
             (tuple_constant
                (length (Str_prefix (S i) σ)) 0)) [].
```

by rewriting according to equivalence_of_repeat_make_tuple_and_sieve, correctness_of_repeat_make_tuple, and equivalence_of_vertical_-and_horizontal_triangle_swap which proves the corollary, and thus the correctness of create_triangle_horizontally and create_triangle_-vertically, with respect to sieve.

We have now finally proved the correctness of all procedures relating to Moessner's sieve and its dual, starting from sieve and create_triangle_-vertically all the way to the helper procedure repeat_make_tuple and our characteristic function rotated_moessner_entry.

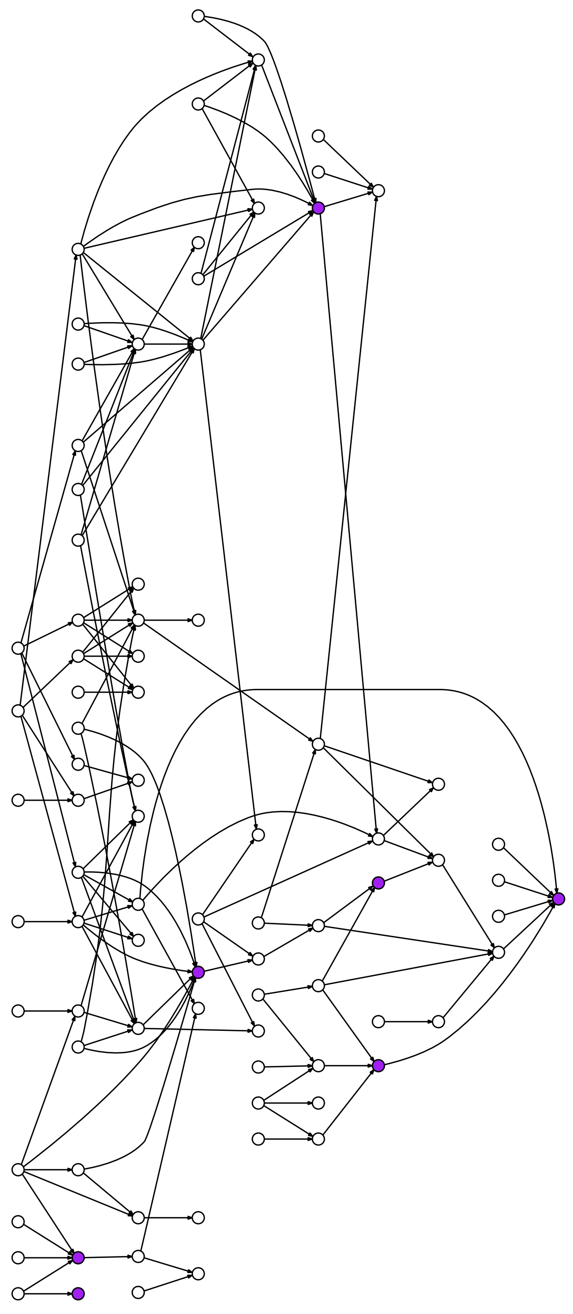
7.5 Summary

In this chapter we have introduced two characteristic functions of Moessner's sieve, moessner_entry and rotated_moessner_entry, which computes the entries of a given Moessner triangle without having to compute the prefix of the sieve. Furthermore, we have presented correctness proofs for the characteristic function rotated_moessner_entry, with respect to create_-triangle_vertically, and for the triangle creation procedure create_-triangle_vertically, with respect to the traditional sieve procedure.

The characteristic functions were derived by observing that every Moessner triangle behaves in a Pascal-like way combined with the fact that the values dropped in the traditional Moessner's sieve enumerates the monomials of the binomial expansion.

The correctness proof of rotated_moessner_entry was done by relating the first column enumerated by rotated_moessner_entry to the partial sums of the monomials of a binomial expansion and by relating the subsequent columns enumerated by rotated_moessner_entry with the repeated application of make_tuple on their prefixes. Furthermore, the correctness proof required the introduction of an auxiliary procedure, repeat_make_tuple, which simplified the mechanics of create_triangle_vertically and thus eased the reasoning needed to complete the proof.

Lastly, the correctness proof of create_triangle_vertically, with respect to sieve, was proved by first establishing an equivalence between sieve_step and make_tuple, followed by the utilization of the procedure repeat_make_tuple to shorten the gap between the operational description of create_triangle_vertically and sieve, thus allowing the completion of the correctness proof.



Dependency graph of the proofs introduced in Chapter 7. Note again a cluster of scaffolding formalizations that results in a sparse graph connecting the individual theorems.

Chapter 8

Proving Moessner's theorem

Every scientist worthy of the name,
but above all a mathematician,
experiences in his work the same sensations as an artist;
his pleasures are as great and of the same nature.

Henri Poincare

The goal of this chapter is to prove Moessner's theorem adapted to the dual sieve, which we call Moessner's idealized theorem. Hence, we prove Moessner's idealized theorem as a corollary of a more general theorem that characterizes the hypotenuse of the nth Moessner triangle created by the dual of Moessner's sieve.

The chapter is structured as follows. In Section 8.1, we characterize the hypotenuse of a Moessner triangle, created with one of our triangle creation procedures, in terms of its seed tuples. Using the characterization of the hypotenuse we move on to characterize the nth triangle created by the dual sieve, in terms of the triangle creation procedure, in Section 8.2. With these proofs, we are able to characterize the hypotenuse of the nth triangle created by the dual of Moessner's sieve, which gives us Moessner's idealized theorem as a corollary.

8.1 Characterizing the hypotenuse of a Moessner triangle

In this section, we first prove an equivalence relation between `monomial` and our characteristic functions, `moessner_entry` and `rotated_moessner_-entry`, which we then use to prove a characterization of the hypotenuse of a Moessner triangle, created with `create_triangle_vertically`, in terms of `monomials`.

8.1.1 Equivalence of moessner-entry and monomial

While we have proved that partially summing the monomials of a binomial expansion yields the first column of a Moessner triangle enumerated by `rotated_moessner_entry`, and likewise proved that partially summing the cth column yields the `(S c)`th column, both expressed in terms of `rotated_-moessner_entry`, we have not yet shown a relation between the hypotenuse of a Moessner triangle and our characteristic functions, `moessner_entry` and `rotated_moessner_entry`.

Now, if we draw the second Moessner triangle created by applying Moessner's sieve of rank 5 on the stream of 1s,

$$\begin{array}{rrrrr} 1 & 1 & 1 & 1 & \mathbf{1} \\ 5 & 6 & 7 & \mathbf{8} & \\ 11 & 17 & \mathbf{24} & & \\ 15 & \mathbf{32} & & & \\ \mathbf{16} & & & & \end{array}$$

we observe that the hypotenuse of the triangle, $(1, 8, 24, 32, 16)$, enumerates the monomials of a binomial expansion, $(1 + t)^r$, when $t = 2$ and $r = 4$. The same expansion is also enumerated by the stream `moessner_entries`,

```
CoFixpoint moessner_entries (r n k t : nat) : Stream nat :=
  (moessner_entry r n k t) :::
  (moessner_entries r n (S k) t).
```

in the opposite order, $(16, 32, 24, 8, 1)$, when given similar arguments. Since this relation can be observed in the general case, we state the following proposition,

```
Corollary moessner_entry_eq_monomial :
  ∀ (n k t : nat),
    k ≤ n →
    moessner_entry n n (n - k) t =
    monomial (S t) n k.
```

between `moessner_entry` and `monomial`, which captures the property above as seen from the two entry indices, `(n - k)` and `n`. The proof follows as a corollary of the equivalent statement of the relation between `rotated_-moessner_entry` and monomial,

```
Theorem rotated_moessner_entry_eq_monomial :
  ∀ (n k t : nat),
    k ≤ n →
    rotated_moessner_entry n k (n - k) t =
    monomial (S t) n k.
```

which is proved using nested induction on the row and column indices, `n` and `k`, along with the following helper theorem,

```
Theorem rotated_moessner_entry_rank_decompose_by_row :
  ∀ (r c n t : nat),
  rotated_moessner_entry (S n) (S r) c t =
  t * rotated_moessner_entry n r c t +
  rotated_moessner_entry n (S r) c t.
```

which captures a new relation between the entries of two Moessner triangles having the same triangle index, `t`, but different ranks, `n` and `(S n)`. We postpone the proof of this theorem to the next chapter, which we devote entirely to this new property, as it opens up a range of new observations and proofs to be made about Moessner's sieve. Lastly, the following modified version of Pascal's rule,

```
Lemma rotated_moessner_entry_constrained_negative_Pascal_s_rule :
  ∀ (n' k' t : nat),
    S k' ≤ n' →
    rotated_moessner_entry n' k' (n' - k') t +
    rotated_moessner_entry n' (S k') (n' - S k') t =
    rotated_moessner_entry n' (S k') (n' - k') t.
```

for `rotated_moessner_entry`, is needed in order to complete the proof of `rotated_moessner_entry_eq_monomial`. The proof follows by nested induction on the row and column indices, `n` and `k`.

As a consequence of having proved `rotated_moessner_entry_eq_monomial`, we are actually in position to state and prove the following simplified version of the binomial theorem,

```
Theorem Binomial_theorem :
  ∀ (t n : nat),
    (S t) ^ n = Str_nth (S n) (stream_partial_sums (monomials t n 0)).
```

which states that the last element of the partial sums of a binomial expansion yields the corresponding exponentiation as stated by the binomial theorem. The proof follows by a series of rewrites using the existing proofs: `stream_partial_sums_acc_monomials_bisim_monomials_sum` and `Str_nth_monomials_sum_eq_rotated_moessner_entry` along with `rotated_moessner_entry_eq_monomial` and the lemma,

```
Lemma monomial_r_eq_n_implies_power :
  ∀ (t r : nat),
    monomial t r r = t ^ r.
```

which is proved by rewriting according to the rules proved for the `binomial_coefficient` function.

8.1.2 The hypotenuse of create-triangle-vertically expressed in terms of monomials

Having proved an equivalence relation between `rotated_moessner_entry` and `monomial`, which describes the individual entries of the hypotenuse of a Moessner triangle, we move on to prove a relation between `create_-triangle_vertically` and `monomials`, which captures the whole hypotenuse of a Moessner triangle.

In order to do so, we once again start by looking at an application example of Moessner's sieve, specifically the first two Moessner triangles of rank 4 and their seed tuples, generated by the dual of Moessner's sieve,

$$\begin{array}{ccccccccccccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 & & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & & & \\
1 & 1 & 1 & 1 & 1 & \mathbf{1} & & & 1 & 1 & 1 & 1 & 1 & \mathbf{1} & \\
0 & 1 & 2 & 3 & \mathbf{4} & & & & 4 & 5 & 6 & 7 & \mathbf{8} & & \\
0 & 1 & 3 & \mathbf{6} & & & & & 6 & 11 & 17 & \mathbf{24} & & & \\
0 & 1 & \mathbf{4} & & & & & & 4 & 15 & \mathbf{32} & & & & \\
0 & \mathbf{1} & & & & & & & 1 & \mathbf{16} & & & & & \\
0 & & & & & & & & 0 & & & & & &
\end{array} \tag{8.1}$$

Here, we want to describe the two hypotenuses, $(1,4,6,4,1)$ and $(16,32,24,8,1)$, in terms of the two vertical seed tuples, $(1,0,0,0,0,0)$ and $(1,4,6,4,1,0)$. Now, we have already observed that the vertical seed tuple of any Moessner triangle enumerates the monomials of the binomial expansion $(1+t)^r$, where t is the triangle index and r is the rank of the triangle. Furthermore, we know that partially summing `monomials` yields `monomials_sum`,

```
Corollary stream_partial_sums_monomials_bisim_monomials_sum :
  ∀ (t r n : nat),
    stream_partial_sums (monomials t r n) ~
    monomials_sum t r n 0.
```

which is equal to the first column of `rotated_moessner_entries`,

```
Corollary rotated_moessner_entries_bisim_monomials_sum :
  ∀ (n t : nat),
    monomials_sum t n 0 0 ~
    rotated_moessner_entries n 0 0 t.
```

Likewise, we also know that the hypotenuse of a given Moessner triangle enumerates the monomials of the same binomial expansion above, $(1+t)^r$, in the opposite order, which is also the case for `moessner_entries`. Thus, if we perform the first application of `make_tuple`, of the `create_triangle_-vertically` procedure, on both triangles in Figure 8.1, we get the following result,

$$\begin{array}{ccccccccccccc}
 & 0 & 0 & 0 & 0 & 0 & & & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & \\
1 & 1 & 1 & 1 & \mathbf{1} & & & 1 & 1 & 1 & 1 & \mathbf{1} & \\
1 & 2 & 3 & \mathbf{4} & & & & 5 & 6 & 7 & \mathbf{8} & & \\
1 & 3 & \mathbf{6} & & & & & 11 & 17 & \mathbf{24} & & & \\
1 & \mathbf{4} & & & & & & 15 & \mathbf{32} & & & & \\
1 & & & & & & & 16 & & & & &
\end{array} \tag{8.2}$$

where the seed tuples correspond to the first columns enumerated by `rotated_moessner_entries`, while the hypotenuses correspond to the streams enumerated by `moessner_entries`, when increasing the start index by 1. If we translate the general case of the state shown in Formula 8.2 to Coq, we get the following theorem,

```
Theorem hypotenuse_create_triangle_vertically_rotated_moessner_entries :
  ∀ (n r c t : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S n) 0)
         (Str_prefix (S n) (rotated_moessner_entries r 0 c t))) =
    Str_prefix n (moessner_entries r (c + n) (S c) t).
```

which captures the relation between the partials sums of a seed tuple, passed to `create_triangle_vertically`, expressed in terms of `rotated_moessner_-entries`, and the tail of the original hypotenuse of the created triangle, expressed in terms of `moessner_entries`. The proof of the above relation is done by induction on the rank, `n`, and rewriting with `make_tuple_rotated_-moessner_entries`, which states that applying `make_tuple` on the `c` column enumerated by `rotated_moessner_entry` yields the `(S c)`th column. Now, if we take one step backwards and state the seed tuple in terms of `monomials`, we get,

```
Theorem hypotenuse_create_triangle_vertically_monomials :
  ∀ (n r t : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S n)) 0)
         (Str_prefix (S (S n)) (monomials t r 0))) =
    (Str_prefix (S n) (moessner_entries r n 0 t)).
```

where `(S c)` has been substituted with `0` and the prefix sizes have been increased by 1. The proof of `hypotenuse_create_triangle_vertically_-monomials` follows from `hypotenuse_create_triangle_vertically_-rotated_moessner_entries` and `rotated_moessner_entries_bisim_-monomials_sum`.

The last piece of the puzzle we need is to define the hypotenuse in terms of `monomials` instead of `moessner_entries`. As already observed, `moessner_-entries` and `monomials` enumerate the same sequence of values in reverse order, when `n = r` in `hypotenuse_create_triangle_vertically_monomials`. Thus, we can state the following theorem,

```
Theorem hypotenuse_create_triangle_vertically :
  ∀ (r t' : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t' r 0))) =
    (rev (Str_prefix (S r) (monomials (S t') r 0))).
```

which captures that passing a list of monomials of the binomial expansion $(1+t)^r$, where $t = t'$, to create_triangle_vertically, returns a Moessner triangle whose hypotenuse is the same binomial expansion, where $t = (S\ t')$.

The proof of hypotenuse_create_triangle_vertically follows from hypotenuse_create_triangle_vertically_monomials and the proof,

```
Lemma rev_Str_prefix_moessner_entries_eq_monomials :
  ∀ (r t' : nat),
    Str_prefix (S r) (moessner_entries r r 0 t') =
    rev (Str_prefix (S r) (monomials (S t') r 0)).
```

stating the reverse relation between monomials and moessner_entries. In order to prove rev_Str_prefix_moessner_entries_eq_monomials, we need a helper procedure,

```
Fixpoint monomials_list (t r n : nat) : list nat :=
  match n with
    | 0 ⇒ [monomial t r 0]
    | S n' ⇒ (monomial t r (S n')) :: (monomials_list t r n')
  end.
```

which enumerates the same values as monomials, but in the opposite order. The role of monomials_list then becomes to connect moessner_entries and monomials by proving a relation that connects the two to monomials_list.

As we have defined monomials_list to enumerate the same monomials as monomials, but in reverse order, we can state the following simple lemma,

```
Lemma monomials_list_eq_rev_Str_prefix_monomials :
  ∀ (n r t : nat),
    monomials_list t r n =
    rev (Str_prefix (S n) (monomials t r 0)).
```

which we prove by induction on the length, n, combined with the helper lemma,

```
Lemma rev_Str_prefix_monomials :
  ∀ (l n r t : nat),
    rev (Str_prefix (S l) (monomials t r n)) =
    monomial t r (n + l) :: rev (Str_prefix l (monomials t r n)).
```

which we prove similarly by induction on the prefix length, l. Having proved the relation between monomials and monomials_list, our last step is to prove the relation between monomials_list and rotated_moessner_entries.

Similar to the lemma above, we can state the following relation between monomials_list and moessner_entries,

```
Lemma monomials_list_eq_Str_prefix_moessner_entries :
  ∀ (l n t' : nat),
    l ≤ n →
    Str_prefix (S l) (moessner_entries n n (n - l) t') =
    monomials_list (S t') n l.
```

for which we are mainly interested in the case where `l = n`, as this covers the whole expansion. The proof of `monomials_list_eq_Str_prefix_moessner_entries` is done by induction on the length, `l`, using the already proved equivalence relation,

```
Corollary moessner_entry_eq_monomial :
  ∀ (n k t : nat),
    k ≤ n →
    moessner_entry n n (n - k) t =
    monomial (S t) n k.
```

for rewriting between the list heads expressed in terms of `moessner_entries` and `monomial`.

By proving `rev_Str_prefix_moessner_entries_eq_monomials`, we get the last piece needed to prove,

```
Theorem hypotenuse_create_triangle_vertically :
  ∀ (r t' : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t' r 0))) =
    (rev (Str_prefix (S r) (monomials (S t') r 0))).
```

which consequently means that we can now reason about any Moessner triangle created by applying `create_triangle_vertically` on a seed tuple expressed in terms of `monomials`.

With this vital proof in hand, our next step is to scale the proof from one Moessner triangle to a list of Moessner triangles, i.e., to prove the relation holds for the *n*th Moessner triangle created by `create_triangles_vertically`, from which Moessner's idealized theorem follows as a corollary.

8.2 Proving Moessner's theorem

Having proved a relation between the seed tuple and hypotenuse of any Moessner triangle created with the triangle creation procedure `create_triangle_vertically`, we move on to describe the *n*th triangle created with `create_triangles_vertically`, in terms of the procedure `create_triangle_vertically` and `monomials`. Being able to characterize the hypotenuse of the *n*th Moessner triangle, we finish the chapter by proving Moessner's idealized theorem as a corollary of this more general theorem.

8.2.1 A list of Moessner triangles

If we return to the theorem `hypotenuse_create_triangle_vertically`, we note that applying `create_triangle_vertically` on `(Str_prefix (S (S r)) (monomials t r 0))`, as the seed tuple, yields the hypotenuse `(rev (Str_prefix (S r) (monomials (S t) r 0)))`, which combined with the fact that `create_triangles_vertically` appends a `0` and reverses the result,

```
Fixpoint create_triangles_vertically (n : nat) (xs ys : tuple)
  : list triangle :=
  match n with
    | 0 ⇒ [create_triangle_vertically xs ys]
    | S n' ⇒
      let ts := create_triangle_vertically xs ys
      in ts :: (create_triangles_vertically n' xs
                 (rev (cons 0 (hypotenuse ts))))
  end.
```

suggests that we can concisely describe the nth triangle as the one created by applying create_triangle_vertically on the seed tuple Str_prefix (S (S r)) (monomials (n + t) r 0),

```
Theorem nth_triangle_create_triangles_vertically :
  ∀ (n r t : nat),
    nth n
      (create_triangles_vertically
         n
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0)))
      [] =
    (create_triangle_vertically
       (tuple_constant (S (S r)) 0)
       (Str_prefix (S (S r)) (monomials (n + t) r 0))).
```

Proving nth_triangle_create_triangles_vertically is done by induction on the triangle index, n, combined with rewriting using the just proved theorem hypotenuse_create_triangle_vertically. Now that we are able to reason about the (n + t)th Moessner triangle created by create_triangles_-vertically, starting from the tth triangle, we have been given a powerful theorem with which we can now prove Moessner's idealized theorem as a corollary.

8.2.2 Moessner's idealized theorem

In order to prove Moessner's idealized theorem, we first prove the following corollary,

```
Corollary bottom_element_of_nth_triangle_is_power :
  ∀ (n r : nat),
    nth 0
        (hypotenuse
           (nth n
                (create_triangles_vertically
                   n
                   (tuple_constant (S (S r)) 0)
                   (Str_prefix (S (S r)) (monomials 0 r 0)))
                []))
        1 = (S n) ^ r.
```

which captures the fact that the bottom-most element of the nth triangle, i.e., the last element of the first column, starting from the triangle $t = 0$, created by our dual sieve, is equal to $(S\ n)^r$, where n is the triangle index and r the rank of the Moessner triangle. The proof follows by rewriting using `nth_triangle_create_triangles_vertically` and `hypotenuse_create_-triangle_vertically` along with `rev_monomials_list_eq_monomials` and `monomial_r_eq_n_implies_power`.

As Moessner's sieve is traditionally modeled as a stream, we take the left-hand side of the proposition in `bottom_element_of_nth_triangle_is_power` and construct a `Stream` which enumerates its result for increasing values of `n`. This gives us the following procedure,

```
CoFixpoint moessner_stream (n r : nat) : Stream nat :=
  (nth 0 (hypotenuse
           (nth n
                (create_triangles_vertically
                   n
                   (tuple_constant (S (S r)) 0)
                   (Str_prefix (S (S r)) (monomials 0 r 0)))
                [])) 1)
    ::: (moessner_stream (S n) r).
```

which enumerates the same values as the result sequence of the traditional version of Moessner's sieve.

If we now define an appropriate stream of successive powers,

```
CoFixpoint successive_powers (b e : nat) : Stream nat :=
  (S b) ^ e ::: successive_powers (S b) e.
```

where `b` is the base and `e` is the exponent, we can state a clear version of Moessner's idealized theorem,

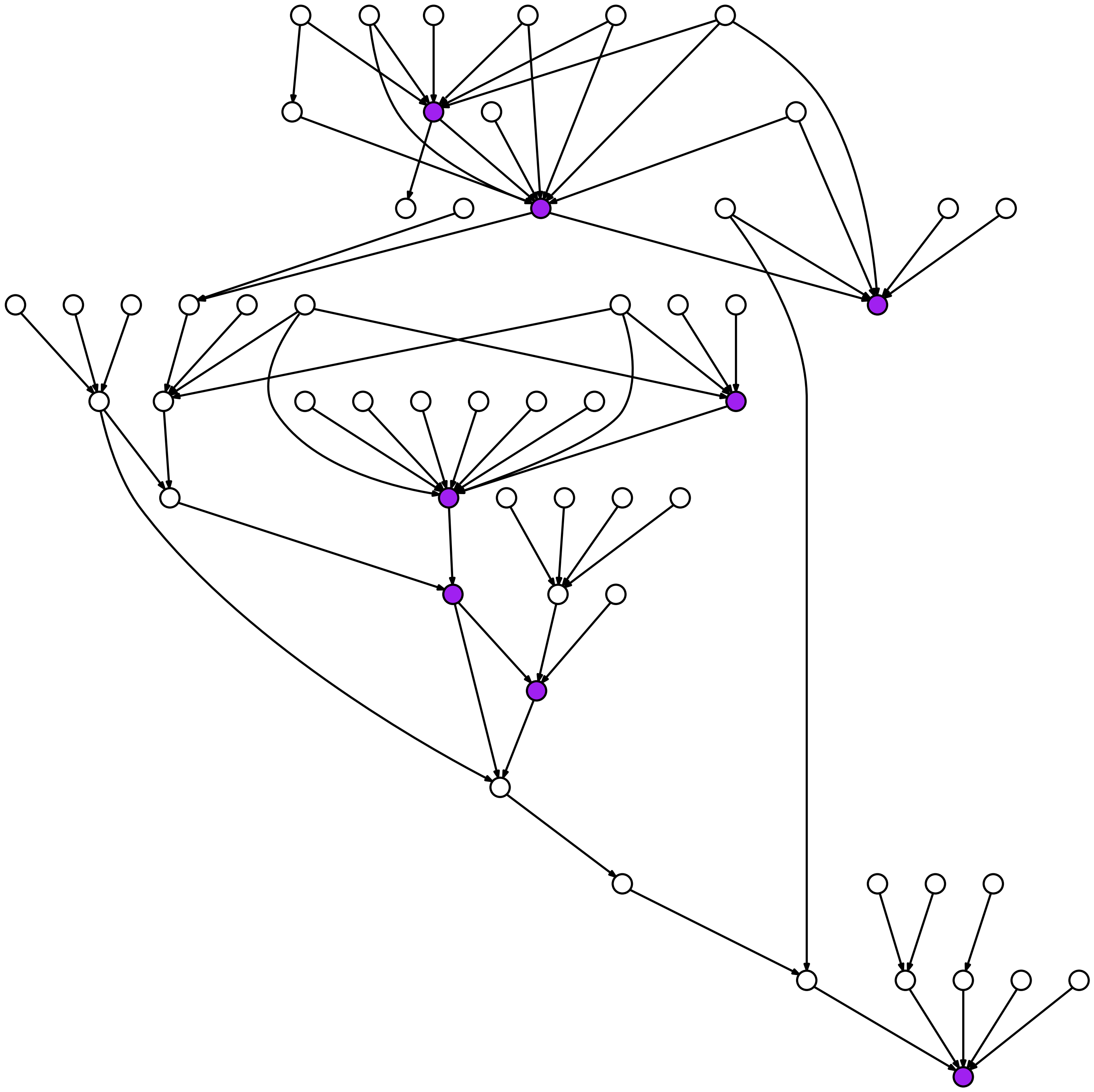
```
Theorem Moessner_s_theorem :
  ∀ (b e : nat),
    moessner_stream b e ∼ successive_powers b e.
```

which follows by coinduction, where we rewrite with `bottom_element_of_-nth_triangle_is_power` in order to prove that the two initial values are equal, while the coinduction hypothesis proves that the stream derivatives are bisimilar. This concludes the proof of Moessner's theorem.

8.3 Summary

In this chapter, we have prove Moessner's theorem adapted to our dual sieve, which we have named Moessner's idealized theorem. Consequently, we proved Moessner's idealized theorem as a corollary of a more general theorem that characterizes the hypotenuse of the `nth` Moessner triangle created by our dual of Moessner's sieve, `create_triangles_vertically`.

The proof of Moessner's idealized theorem was done by first proving an equivalence between `rotated_moessner_entry` and `monomial`, which led to the characterization of the hypotenuse of a Moessner triangle created by `create_triangle_vertically`, expressed in terms of the `monomials` procedure. By extending the characterization to describe the *n*th triangle created by `create_triangles_vertically` in terms of `create_triangle_vertically` and `monomials`, we obtained a more general proof characterizing the output of Moessner's sieve from which Moessner's idealized theorem then followed as a corollary.



Dependency graph of the proofs introduced in Chapter 8. Notice the clear flow from top to bottom of the theorems driving the chapter, each surrounded by a small set of helper lemmas.

Chapter 9

A grid of triangles

The trick, William Potter,
is not minding that it hurts.

Robert Bolt, Lawrence of Arabia

However vast the darkness,
we must supply our own light.

Stanley Kubrick

In this chapter, we introduce a new combinatorial property which connects Moessner triangles of different rank but with the same triangle index, thus acting as a dual to the existing connection between Moessner triangles of the same rank but different triangle index. This duality proposes the view of Moessner's sieve as generating a 2-dimensional grid of triangles instead of just a 1-dimensional sequence of triangles. Specifically, we first introduce a rank-upgrading procedure which takes a seed tuple of a Moessner triangle of rank n and returns the seed tuple of the same Moessner triangle of rank $(S\ n)$. Furthermore, we also prove several rank decomposition rules, which describe an entry of a Moessner triangle of rank $(S\ n)$ as a sum of entries in the same Moessner triangle of rank n.

The chapter is structured as follows. In Section 9.1, we motivate the idea of viewing Moessner's sieve as generating a grid of triangles, and introduce a rank-upgrading procedure, which takes a seed tuple of a Moessner triangle of rank n and returns the seed tuple of the same Moessner triangle of rank $(S\ n)$. Furthermore, we also prove the correctness of the rank-upgrading procedure and demonstrate its use. As a dual to the first section, we introduce a set of rank decomposition rules in Section 9.2, which allows us to describe any entry of a Moessner triangle of rank $(S\ n)$ as a sum of entries in the same Moessner triangle of rank n. We prove the decomposition rules in terms of our characteristic functions, `moessner_entry` and `rotated_moessner_entry`, and obtain the equivalent rules for `create_triangle_vertically` as corollaries.

9.1 Generating a grid of triangles with Moessner's sieve

In this section, we propose the idea of viewing the output of Moessner's sieve as a grid of triangles by first observing a connection between the seed tuples of the tth Moessner triangle of different rank, n and $(S\ n)$. Using this observation, we introduce a rank-upgrading procedure, `upgrade_seed_tuple`, which takes a seed tuple of a Moessner triangle of rank n and returns the seed tuple of the same Moessner triangle of rank $(S\ n)$. Furthermore, we prove the correctness of the `upgrade_seed_tuple` procedure and demonstrate its application.

9.1.1 A connection between seed tuples

In order to motivate the idea of Moessner's sieve generating a grid of triangles, we start by examining the first three Moessner triangles of rank 3 and 4, along with their respective seed tuples,

```
     0  0  0  0  0          0    0    0  0  0            0     0    0    0  0

1    1  1  1  1        1    1    1    1  1          1    1     1    1    1
0    1  2  3           3    4    5    6             6    7     8    9
0    1  3              3    7   12                 12   19    27
0    1                 1    8                       8   27
0                      0                            0


     0  0  0  0  0  0       0    0    0  0  0  0     0     0    0    0  0  0

1    1  1  1  1  1     1    1    1    1  1  1       1    1     1    1    1  1
0    1  2  3  4        4    5    6    7  8          8    9    10   11   12
0    1  3  6           6   11   17   24            24   33    43   54
0    1  4              4   15   32                 32   65   108
0    1                 1   16                      16   81
0                      0                            0
```

Now, for both sieves we know that we can move from left to right, i.e., increase the index of the triangles, but we do not know if we can move from top to bottom, i.e., increase the rank of the triangles. However, if we remember that we can characterize each seed tuple as an instance of the binomial expansion $(1+t)^n$, where n is the rank of the Moessner triangle and t is the triangle index, we search for a combinatorial property that allows us to go from the seed tuple corresponding to the binomial expansion where $n = n'$ to the seed tuple corresponding to the binomial expansion where $n = (S\ n')$, thus obtaining the needed vertical movement in the grid of triangles.

If we examine the two seed tuples generated by the first Moessner triangles, $(1, 3, 3, 1)$ and $(1, 4, 6, 4, 1)$, we observe that we can obtain the second seed tuple from the first using the following scheme,

$$\begin{aligned} 1 &= 1 + 0 \\ 4 &= 3 + 1 \\ 6 &= 3 + 3 \\ 4 &= 1 + 3 \\ 1 &= 0 + 1, \end{aligned} \tag{9.1}$$

where we obtain the $(S\ i)$th element of rank $(S\ n)$ by adding the $(S\ i)$th element of rank n plus the value of an accumulator which contains the value of the ith element of rank n – coincidentally this calculation is also equivalent to an application of Pascal's rule in Pascal's triangle for these values. However, when we examine the next pair of seed tuples, $(1, 6, 12, 8)$ and $(1, 8, 24, 32, 16)$, we realize that the above scheme is insufficient for calculating the second tuple from the first. Fortunately, we receive a hint from the fact that the last elements of the two tuples are equal to 2^3 and 2^4, respectively, which means that we can obtain the latter by multiplying the former by 2. With this in mind, we change the scheme accordingly and get,

$$\begin{aligned} 16 &= 2 \cdot 8 + 0 \\ 32 &= 2 \cdot 12 + 8 \\ 24 &= 2 \cdot 6 + 12 \\ 8 &= 2 \cdot 1 + 6 \\ 1 &= 2 \cdot 0 + 1, \end{aligned} \tag{9.2}$$

which now yields the desired result. It turns out that this Pascal-like property, of adding the two nearest entries of the seed tuple of rank n', holds in general if we substitute the 2 with $(1 + t)$. For example, if we look at the hypotenuses of the third pair of triangles, where $t = 2$, we get the following calculations,

$$\begin{aligned} 81 &= (1 + 2) \cdot 27 + 0 \\ 108 &= (1 + 2) \cdot 27 + 27 \\ 54 &= (1 + 2) \cdot 9 + 27 \\ 12 &= (1 + 2) \cdot 1 + 9 \\ 1 &= (1 + 2) \cdot 0 + 1, \end{aligned} \tag{9.3}$$

which confirm the correctness of the formula – this property can also be seen from the multiplicative property, $(1 + t)^{1+n} = (1 + t) \cdot (1 + t)^n$, of the binomial expansion. Thus, we have now demonstrated how to obtain the seed tuple of rank $(S\ n)$, when given the seed tuple of rank n, which means that we can now move in a vertical direction as well as a horizontal direction in the grid of triangles shown at the beginning of this section.

Having covered the motivation for perceiving Moessner's sieve as generating a grid of triangles, rather than a sequence of triangles, we move on to construct a rank-upgrading procedure, which given a seed tuple of rank n returns the corresponding seed tuple of rank $(S\ n)$, thus implementing the vertical direction discussed above.

9.1.2 Rank upgrading procedure

When taking the description of the rank-upgrading procedure in the previous section and translating it into Coq, we initially note that the procedure should take a seed tuple, `xs`, an accumulator, `a`, and a triangle index, `t`, as inputs. Furthermore, we want to pattern match on the structure of the seed tuple, `xs`, as the procedure works by traversing the tuple and operating on its elements. Lastly, we observe that for the base case of the pattern matching, `xs = []`, we simply return a list containing just the accumulator, while in the inductive case of the pattern matching, `xs = x :: xs'`, we add the accumulator, `a`, to `(S t) * x` and `cons` the intermediate result with the result of the recursive call on `xs'`. Putting these pieces together we get the procedure,

```
Fixpoint upgrade_seed_tuple_aux (t a : nat) (xs : tuple) : tuple :=
  match xs with
    | [] ⇒ [a]
    | x :: xs' ⇒ (S t) * x + a :: upgrade_seed_tuple_aux t x xs'
  end.
```

for which we also define a function that initializes the accumulator to `0`,

```
Definition upgrade_seed_tuple (t : nat) (xs : tuple) : tuple :=
  upgrade_seed_tuple_aux t 0 xs.
```

such that the three examples in Figure 9.1-9.3 can be expressed as the propositions,

```
upgrade_seed_tuple 0 (rev (Str_prefix 4 (monomials 1 3 0))) =
rev (Str_prefix 5 (monomials 1 4 0)).

upgrade_seed_tuple 1 (rev (Str_prefix 4 (monomials 2 3 0))) =
rev (Str_prefix 5 (monomials 2 4 0)).

upgrade_seed_tuple 2 (rev (Str_prefix 4 (monomials 3 3 0))) =
rev (Str_prefix 5 (monomials 3 4 0)).
```

where the presence of `rev` is the result of the procedure having to read the seed tuples from the bottom up. As in the case of our previous formalizations, we now proceed by proving the correctness of our rank-upgrading procedure, `upgrade_seed_tuple`, which corresponds to proving a generalization of the three example propositions above.

9.1.3 Correctness of rank upgrading procedure

To prove the correctness of our newly defined rank-upgrading procedure, `upgrade_seed_tuple`, we use the knowledge of the previous section to characterize the input and output of the procedure in terms of the `monomials` function, which gives the following correctness proof,

```
Theorem correctness_of_upgrade_seed_tuple :
  ∀ (n t : nat),
    upgrade_seed_tuple t (rev (Str_prefix (S n) (monomials (S t) n 0))) =
    rev (Str_prefix (S (S n)) (monomials (S t) (S n) 0)).
```

which captures the relation between two seed tuples of rank `n` and `(S n)`.

In order to prove `correctness_of_upgrade_seed_tuple`, we restate the proof by first replacing the occurrences of `monomials` with `moessner_entries`, which is done by rewriting with the already proved lemma,

```
Lemma rev_Str_prefix_moessner_entries_eq_monomials :
  ∀ (r t' : nat),
    Str_prefix (S r) (moessner_entries r r 0 t') =
    rev (Str_prefix (S r) (monomials (S t') r 0)).
```

followed by unfolding the definition of `upgrade_seed_tuple` which yields the proposition,

```
   S t * moessner_entry n n 0 t
   :: upgrade_seed_tuple_aux t (moessner_entry n n 0 t)
        (Str_prefix n (moessner_entries n n 1 t)) =
   Str_prefix (S (S n)) (moessner_entries (S n) (S n) 0 t)
```

that we subsequently turn into the correctness theorem of the underlying rank-upgrading procedure `upgrade_seed_tuple_aux`,

```
Theorem correctness_of_upgrade_seed_tuple_aux :
  ∀ (n t : nat),
    (S t) * (moessner_entry n n 0 t)
             :: upgrade_seed_tuple_aux t (moessner_entry n n 0 t)
             (Str_prefix n (moessner_entries n n 1 t)) =
    Str_prefix (S (S n)) (moessner_entries (S n) (S n) 0 t).
```

However, proving the correctness of `upgrade_seed_tuple_aux` requires us to once again state the proof a bit differently. As such, we replace `moessner_-entries` with `monomials_list` by rewriting with,

```
Lemma monomials_list_eq_Str_prefix_moessner_entries :
  ∀ (l n t' : nat),
    l ≤ n →
    Str_prefix (S l) (moessner_entries n n (n - l) t') =
    monomials_list (S t') n l.
```

where `l = n`, giving us a new correctness proof of `upgrade_seed_tuple_aux`,

```
Lemma correctness_of_upgrade_seed_tuple_aux_list :
  ∀ (n r t : nat),
    upgrade_seed_tuple_aux
      t (monomial (S t) r (S n)) (monomials_list (S t) r n) =
    monomials_list (S t) (S r) (S n).
```

with respect to `monomials_list`,

```
Fixpoint monomials_list (t r n : nat) : list nat :=
  match n with
    | 0 ⇒ [monomial t r 0]
    | S n' ⇒ (monomial t r (S n')) :: (monomials_list t r n')
  end.
```

and parameterized over both the rank and entry indices, `n` and `r`, instead of just the rank, `n`. Having stated a correctness proof of `upgrade_seed_tuple_-aux`, where `n` and `r` are independent, we can now prove it by induction on the rank, `n`, and rewriting according to,

```
Theorem monomial_decompose_rank :
  ∀ (t r' n' : nat),
    monomial t (S r') (S n') =
    monomial t r' (S n') + t * monomial t r' n'.
```

which captures the Pascal-like decomposition rule of the monomials in the seed tuples, demonstrated in the previous section. The proof of `monomial_-decompose_rank` follows from Pascal's rule,

```
Theorem Pascal_s_rule' :
  ∀ (n' k' : nat),
    C(S n', S k') = C(n', S k') + C(n', k').
```

thus proving the correctness of `upgrade_seed_tuple_aux` in terms of `monomials_list`.

Having proved `correctness_of_upgrade_seed_tuple_aux_list`, we can then prove `correctness_of_upgrade_seed_tuple_aux` by case analysis on the rank, `n`, and rewriting using the properties of the `monomial` function. Finally, we obtain the proof of `correctness_of_upgrade_seed_tuple` as we have already reduced the proof to a proposition stated in terms of `upgrade_-seed_tuple_aux`.

Lastly, as a corollary of the correctness proofs, we can prove that applying `upgrade_seed_tuple` on the hypotenuse of the `t`th Moessner triangle of rank `n` is equivalent to the hypotenuse of the `t`th Moessner triangle of rank `(S n)`,

```
Corollary upgrade_seed_tuple_create_triangle_vertically :
  ∀ (n t : nat),
    upgrade_seed_tuple
      t (hypotenuse
           (create_triangle_vertically
              (tuple_constant (S (S n)) 0)
              (Str_prefix (S (S n)) (monomials t n 0)))) =
  hypotenuse
    (create_triangle_vertically
       (tuple_constant (S (S (S n))) 0)
       (Str_prefix (S (S (S n))) (monomials t (S n) 0))).
```

thus demonstrating how the procedure `upgrade_seed_tuple` can be used to switch the rank between calls to `create_triangle_vertically`.

Having proved the correctness of `upgrade_seed_tuple` and demonstrated its use, we take a step back and investigate the dual of this section. Specifically, our next step is to prove how to decompose the entries of the *t*th Moessner triangle of rank $(S\ n)$ in terms of the same Moessner triangle of rank n.

9.2 Rank decomposition of Moessner triangles

In this section, we take the dual approach of the previous section by first motivating the introduction of a series of rank decomposition rules, which allows us to describe the entries of a Moessner triangle of rank $(S\ n)$ in terms of the same Moessner triangle of rank n. Having covered the motivation for the rank decomposition rules, we then formalize them in Coq using our characteristic function, `rotated_moessner_entry`, and afterwards adapt the rules to our triangle creation procedure, `create_triangle_vertically`, as corollaries.

9.2.1 Motivating the decomposition of Moessner triangles

Starting from the same example as in the previous section, we examine the first three Moessner triangles of rank 3 and 4,

	0	0	0	0	0
1	1	1	1	1	
0	1	2	3		
0	1	3			
0	1				
0					

	0	0	0	0	0
1	1	1	1	1	
3	4	5	6		
3	7	12			
1	8				
0					

	0	0	0	0	0
1	1	1	1	1	
6	7	8	9		
12	19	27			
8	27				
0					

	0	0	0	0	0	0
1	1	1	1	1	1	
0	1	2	3	4		
0	1	3	6			
0	1	4				
0	1					
0						

	0	0	0	0	0	0
1	1	1	1	1	1	
4	5	6	7	8		
6	11	17	24			
4	15	32				
1	16					
0						

	0	0	0	0	0	0
1	1	1	1	1	1	
8	9	10	11	12		
24	33	43	54			
32	65	108				
16	81					
0						

and use the knowledge we have gathered so far to drive our motivation. Instead of looking at the calculations in Formula 9.2 and 9.3 as the upgrading of a seed tuple, we flip the perspective and see it as an example of decomposing the hypotenuse in terms of the Moessner triangle of lower rank. Taking this idea one step further, we propose the idea that there exists a set of rank decomposition rules which work for all entries of a triangle and not just the hypotenuse/seed tuple. With this idea in mind, we focus on the first column of the second and third pair of Moessner triangles and try to apply the same scheme as before, except that we make two minor adjustments,

1. we multiply the first term with t instead of $(1+t)$, and
2. we start with an accumulator equal to the last value of the column instead of 0,

which gives us the following calculations, for the second and third triangles,

$$\begin{aligned} 16 &= 1\cdot 8 + 8 \\ 15 &= 1\cdot 7 + 8 \\ 11 &= 1\cdot 4 + 7 \\ 5 &= 1\cdot 1 + 4 \\ 1 &= 1\cdot 0 + 1, \end{aligned} \qquad \text{and} \qquad \begin{aligned} 81 &= 2\cdot 27 + 27 \\ 65 &= 2\cdot 19 + 27 \\ 33 &= 2\cdot 7 + 19 \\ 9 &= 2\cdot 1 + 7 \\ 1 &= 2\cdot 0 + 1, \end{aligned} \tag{9.4}$$

demonstrating that the property also holds for the initial column of every Moessner triangle. Remembering that the different Moessner triangles are constructed using Pascal's rule, we restate the calculations in Formula 9.4 as,

$$\begin{aligned} 16 &= 2\cdot 8 + 0 \\ 15 &= 2\cdot 7 + 1 \\ 11 &= 2\cdot 4 + 3 \\ 5 &= 2\cdot 1 + 3 \\ 1 &= 2\cdot 0 + 1, \end{aligned} \qquad \text{and} \qquad \begin{aligned} 81 &= 3\cdot 27 + 0 \\ 65 &= 3\cdot 19 + 8 \\ 33 &= 3\cdot 7 + 12 \\ 9 &= 3\cdot 1 + 6 \\ 1 &= 3\cdot 0 + 1, \end{aligned} \tag{9.5}$$

by realizing that each of the values used for accumulators, in Formula 9.4, is actually the sum of one of the values in the seed tuple (western neighbor) and the entry which we have already multiplied by t (northern neighbor),

$$\begin{aligned} 16 &= 1\cdot 8 + (8+0) \\ 15 &= 1\cdot 7 + (7+1) \\ 11 &= 1\cdot 4 + (4+3) \\ 5 &= 1\cdot 1 + (1+3) \\ 1 &= 1\cdot 0 + (0+1), \end{aligned} \qquad \text{and} \qquad \begin{aligned} 81 &= 2\cdot 27 + (27+0) \\ 65 &= 2\cdot 19 + (19+8) \\ 33 &= 2\cdot 7 + (7+12) \\ 9 &= 2\cdot 1 + (1+6) \\ 1 &= 2\cdot 0 + (0+1). \end{aligned}$$

Thus, we get $(1+t)$ times the entry above the desired entry (northern neighbor) and a value of the seed tuple/hypotenuse of the previous triangle (western neighbor).

Noting that we now have a Pascal-like rule which works across ranks, we examine whether it also holds true for the subsequent columns of the Moessner triangles. As such, we try to calculate the second column of the second and third pair of triangles using the first columns for accumulator values, instead of the seed tuples,

$$\begin{aligned} 32 &= 2 \cdot 12 + 8 \\ 17 &= 2 \cdot 5 + 7 \\ 6 &= 2 \cdot 1 + 4 \\ 1 &= 2 \cdot 0 + 1, \end{aligned} \qquad \text{and} \qquad \begin{aligned} 108 &= 3 \cdot 27 + 27 \\ 43 &= 3 \cdot 8 + 19 \\ 10 &= 3 \cdot 1 + 7 \\ 1 &= 3 \cdot 0 + 1. \end{aligned} \tag{9.6}$$

Again, we obtain the desired results, which demonstrates a consistent Pascal-like property across ranks and triangles. Thus, we have now shown how it is possible to state an entry of a Moessner triangle of rank $(S\ n)$ as a sum of entries in the same Moessner triangle of rank n.

Next, we transform our motivating examples into concrete proofs of the rank decomposition rules in the Coq proof assistant.

9.2.2 Formalizing the decomposition rules in Coq

A subtle point lies in the fact that while the Moessner triangles have a finite number of entries in each column, this is not the case of our characteristic function `rotated_moessner_entry`,

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	5	6	7	8	9	9	10	11	12	13
1	3	6	10	15	11	17	24	33	42	33	43	54	66	76
1	4	10	20	35	15	32	55	88	130	65	108	162	192	268
1	5	15	35	70	16	48	103	191	321	81	189	351	543	811

as the gray values above are the results of computing entries outside of the Moessner triangles using our characteristic function. Thus, we obtain a more general, and easier to state, proof of the rank decomposition rules by proving the property for the characteristic function, `rotated_moessner_entry`, rather than directly on the triangle creation procedure, `create_triangle_-vertically`, or on the simplified procedure, `repeat_make_tuple`.

In the previous section, we demonstrated two Pascal-like properties that could be merged into one simpler property, expressing an entry of a Moessner triangle of rank $(S\ n)$ in terms of the same entry in the triangle of rank n along with the entry above it (northern neighbor), which works for all columns of a Moessner triangle. Consequently, we start by stating and proving this last rank decomposition rule,

```
Theorem rotated_moessner_entry_rank_decompose_by_row :
  ∀ (r c n t : nat),
  rotated_moessner_entry (S n) (S r) c t =
  t * rotated_moessner_entry n r c t +
  rotated_moessner_entry n (S r) c t.
```

which states that the entry in the $(S\ r)$th row and cth column of a Moessner triangle of rank $(S\ n)$, is the sum of t times the entry at the rth row and cth column of rank n and the entry at the $(S\ r)$th row and cth column of rank n. As mentioned, this rule captures the examples we have shown above, and the proof of the theorem proceeds by nested induction on the row and column indices, r and c, and rewriting according to already proved properties of rotated_moessner_entry and monomial. From this theorem follows the equivalent proof for moessner_entry as a corollary,

```
Corollary moessner_entry_rank_decompose_by_row :
  ∀ (r c n t : nat),
   moessner_entry (S n) (c + S r) c t =
   t * moessner_entry n (c + r) c t +
   moessner_entry n (c + S r) c t.
```

along with the two Pascal-like properties,

```
Corollary rotated_moessner_entry_rank_decompose_Pascal_like_c_eq_0 :
  ∀ (r n t : nat),
    rotated_moessner_entry (S n) (S r) 0 t =
    (S t) * rotated_moessner_entry n r 0 t +
    monomial t n (S r).
```

and

```
Corollary rotated_moessner_entry_rank_decompose_Pascal_like_c_gt_0 :
  ∀ (r c n t : nat),
  rotated_moessner_entry (S n) (S r) (S c) t =
  (S t) * rotated_moessner_entry n r (S c) t +
  rotated_moessner_entry n (S r) c t.
```

and their equivalent proofs for moessner_entry,

```
Corollary moessner_entry_rank_decompose_Pascal_like_c_eq_0 :
  ∀ (r n t : nat),
    moessner_entry (S n) (S r) 0 t =
    S t * moessner_entry n r 0 t +
    monomial t n (S r).
```

and

```
Corollary moessner_entry_rank_decompose_Pascal_like_c_gt_0 :
  ∀ (r c n t : nat),
   moessner_entry (S n) (S c + S r) (S c) t =
   (S t) * moessner_entry n (S c + r) (S c) t +
   moessner_entry n (c + S r) c t.
```

Lastly, we can translate these rank decomposition rules to work on create_-triangle_vertically, which gives the following three corollaries,

```
Corollary create_triangle_vertically_decompose_by_rank :
  ∀ (i j r t : nat),
    j ≤ r → S i ≤ r - j →
    (nth (S i) (nth j
      (create_triangle_vertically
        (tuple_constant (S (S (S r))) 0)
        (Str_prefix (S (S (S r))) (monomials t (S r) 0))) []) 0) =
    t * (nth i (nth j
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0) +
    (nth (S i) (nth j
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0).
```

and

```
Corollary create_triangle_vertically_rank_decompose_Pascal_like_c_eq_0 :
  ∀ (i r t : nat),
    i < (S r) →
    (nth (S i) (nth 0
      (create_triangle_vertically
         (tuple_constant (S (S (S r))) 0)
         (Str_prefix (S (S (S r))) (monomials t (S r) 0))) []) 0) =
    (S t) * (nth i (nth 0
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0) +
    monomial t r (S i).
```

and

```
Corollary create_triangle_vertically_rank_decompose_Pascal_like_c_gt_0 :
  ∀ (i j r t : nat),
    j ≤ r → S i ≤ r - j →
    (nth (S i) (nth (S j)
      (create_triangle_vertically
         (tuple_constant (S (S (S r))) 0)
         (Str_prefix (S (S (S r))) (monomials t (S r) 0))) []) 0) =
    (S t) * (nth i (nth (S j)
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0) +
    (nth (S i) (nth j
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0).
```

where the complexity of the statements once again emphasizes the gains we receive by reasoning with our characteristic function rather than directly on the triangle creation procedures, `create_triangle_horizontally` and `create_triangle_vertically`.

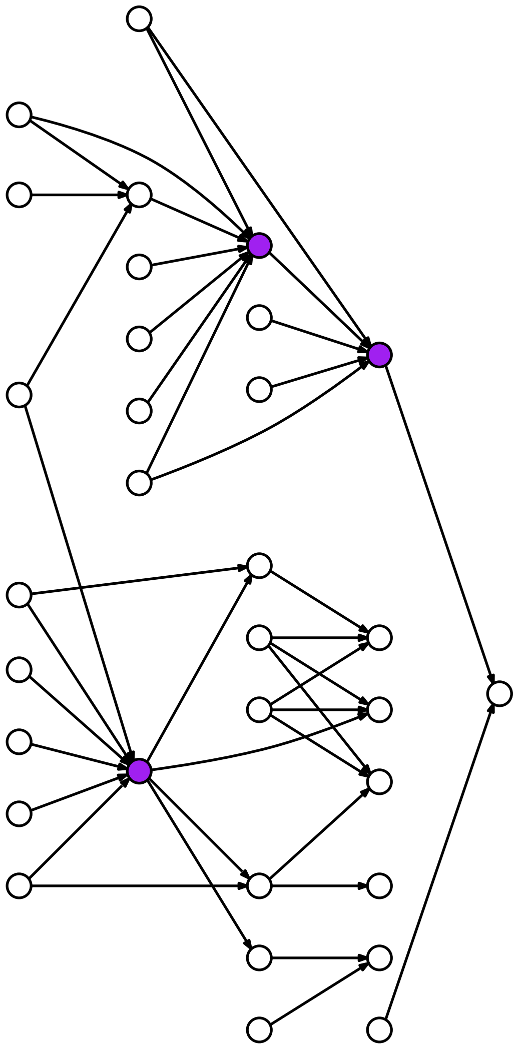
Combining the above proofs and the procedure of the previous section, we have now introduced and proved a new property of Moessner's sieve that creates a vertical connection between the seed tuples and entries of two Moessner triangles with the same triangle index, t, but different ranks, n and $(S\ n)$, thus acting as a dual to the existing properties which horizontally connects two triangles with different triangle index, t, but same rank, n, in this implicit grid of triangles.

9.3 Summary

In this chapter, we have introduced a new combinatorial property which connects Moessner triangles of different rank but with the same triangle index, thus acting as a dual to the existing connection between Moessner triangles of the same rank but different triangle index. This duality implies a 2-dimensional grid of Moessner triangles, where the triangle index is increasing as we go along the horizontal axis, from left to right, while the rank is increasing when going along the vertical axis, from top to bottom. These grid properties have been introduced as a rank-upgrading procedure, which takes a seed tuple of the tth Moessner triangle of rank n and returns the seed tuple of the tth Moessner triangle of rank $(S\ n)$, and several rank decomposition rules, which describe an entry of the tth Moessner triangle of rank $(S\ n)$ as a sum of entries in the tth Moessner triangle of rank n.

The rank-upgrading procedure, `upgrade_seed_tuple`, was the result of the observation that we could obtain the seed tuple of the Moessner triangle of rank $(S\ n)$ by adding pairs of entries in the seed tuple of the Moessner triangle of rank n where one was multiplied with the triangle index.

Conversely, the rank decomposition rules were the result of exploring whether the decomposition rule only applied for the seed tuples or if it persisted into the entries of the Moessner triangles.



Dependency graph of the proofs introduced in Chapter 9. Notice again the sparsity and size of the graph, reflecting the robustness of the scaffolding to obtain new results.

Chapter 10

Proving Long's theorem

One chord is fine.
Two chords are pushing it.
Three chords and you're into jazz.

Lou Reed

Simple things should be simple,
complex things should be possible.

Alan Kay

The goal of this chapter is to prove an idealized version of Long's theorem stated in terms of the dual of Moessner's sieve. Thus, we first adapt Long's original theorem to the dual sieve that leads us to state two versions of Long's theorem, the first of which generalizes the seed value of 1 in Moessner's idealized theorem to an arbitrary constant, d, which we call Long's weak theorem, while the second generalizes Long's weak theorem from a seed tuple of one nonzero entry, d, to two, c and d, which we call Long's idealized theorem. Lastly, we state a conjecture which generalizes Long's idealized theorem from a seed tuple with two nonzero entries, c and d, to a seed tuple with an arbitrary number of nonzero entries.

The chapter is structured as follows. In Section 10.1, we start by adapting Long's theorem to the dual sieve, which motivates the statement of Long's weak theorem and Long's idealized theorem. As a result, we prove Long's weak theorem in Section 10.2, which generalizes Moessner's idealized theorem from a seed value of 1 to an arbitrary constant d, and prove Long's idealized theorem in Section 10.3, which generalizes the weak form of Long's theorem from a seed tuple of one nonzero entry, d, to two, c and d. Finally, we conjecture the generalization of Long's idealized theorem from a seed tuple with two nonzero entries, c and d, to a seed tuple with an arbitrary number of nonzero entries, in Section 10.4.

10.1 Long's theorem and the dual of Moessner's sieve

In order to prove Long's theorem using our dual sieve, we start by repeating its traditional definition and afterwards adapt it to the dual sieve.

Long's original theorem states that if we apply Moessner's sieve of rank k on an initial sequence which can be described as an arithmetic progression,

$$c, c+d, c+2d, c+3d, \dots,$$

we obtain the result sequence,

$$c \cdot 1^{k-1}, (c+d) \cdot 2^{k-1}, (c+2d) \cdot 3^{k-1}, \dots,$$

which we can visualize as the sieve,

$$\begin{array}{cccccccccc}
c & c+d & c+2d & c+3d & c+4d & c+5d & c+6d & c+7d & \dots \\
c & 2c+d & 3c+3d & & 4c+7d & 5c+12d & 6c+18d & & \dots \\
c & 3c+d & & & 7c+8d & 12c+20d & & & \dots \\
c & & & & 8c+8d & & & & \dots
\end{array}$$

Furthermore, Long [22] also noted that we can generalize the sieve above by adding a row of ds,

$$\begin{array}{cccccccccc}
d & d & d & d & d & d & d & d & d & d \\
c & c+d & c+2d & c+3d & & c+4d & c+5d & c+6d & c+7d & \\
c & 2c+d & 3c+3d & & & 4c+7d & 5c+12d & 6c+18d & & \\
c & 3c+d & & & & 7c+8d & 12c+20d & & & \\
c & & & & & 8c+8d & & & &
\end{array}$$

which unfortunately gives us an inconsistent initial column, since it contains a d at the top but cs in the remaining entries. So, we take the liberty of adjusting the initial configuration of the sieve to better suit our dual sieve, by ridding ourselves of the above inconsistency. Thus, we move the cs of the initial column into the vertical seed tuple, and at the same time generalize to a single seed value c, while putting the ds in the horizontal seed tuples, yielding the following sieve,[1]

$$\begin{array}{c|r|r|r|r|r|r|r|r|r|r}
 & d & d & d & d & d & d & d & d & d & d \\
 & & & & & & & & & & \\
c & c+d & c+2d & c+3d & c+4d & & c+5d & c+6d & c+7d & c+8d & \\
0 & c+d & 2c+3d & 3c+6d & & & 4c+11d & 5c+17d & 6c+24d & & \\
0 & c+d & 3c+4d & & & & 7c+15d & 12c+32d & & & \\
0 & c+d & & & & & 8c+16d & & & & \\
0 & & & & & & & & & &
\end{array}$$

While moving the cs has changed the coefficients of the ds in the sieve, we now have a more consistent initial configuration, which we believe to be in

[1]The addition of the dotted vertical lines between each column is for the sake of readability.

the spirit of Long's original theorem, with one constant, c, in the vertical seed tuple and the horizontal seed tuples filled with the constant d. As it turns out, we can perform a further generalization of the initial configuration by replacing the sequence of ds with a d followed by 0s, while putting it in the vertical seed tuple, as we did with the sequence of 1s when we defined the dual of Moessner's sieve in Chapter 5,

$$
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & \\
d & d & d & d & d & d & & d & d & d & d & d & \\
c & c+d & c+2d & c+3d & c+4d & & & c+5d & c+6d & c+7d & c+8d & & \\
0 & c+d & 2c+3d & 3c+6d & & & & 4c+11d & 5c+17d & 6c+24d & & & \\
0 & c+d & 3c+4d & & & & & 7c+15d & 12c+32d & & & & \\
0 & c+d & & & & & & 8c+16d & & & & & \\
0 & & & & & & & & & & & &
\end{array}
$$

This results in a minimal initial configuration consisting of a vertical seed tuple containing a constant d and a constant c followed by 0s. We can now state Long's idealized theorem as starting from an initial configuration with a vertical seed tuple of length k, where $k \geq 2$, and containing a c and a d, which yields the result sequence enumerated by the formula,

$$d \cdot (1+t)^{k-2} + c \cdot (1+t)^{k-3}. \tag{10.1}$$

As a result of the transformations made above, we now notice that the coefficients of the cs correspond to the values of Moessner triangles at rank 3 while the coefficients of the ds now correspond to the values of Moessner triangles at rank 4. This observation suggests that we can view the above sieve as the composition of two sieves, one creating Moessner triangles of rank 3 filled with cs,

$$
\begin{array}{cccccccccccc}
 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & \\
c & c & c & c & c & & c & c & c & c & c & \\
0 & c & 2c & 3c & & & 3c & 4c & 5c & 6c & & \\
0 & c & 3c & & & & 3c & 7c & 12c & & & \\
0 & c & & & & & c & 8c & & & & \\
0 & & & & & & 0 & & & & &
\end{array}
$$

and one creating Moessner triangles of rank 4 filled with ds,

$$
\begin{array}{cccccccccccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & & \\
d & d & d & d & d & d & & d & d & d & d & d & d & \\
0 & d & 2d & 1d & 4d & & & 4d & 5d & 6d & 7d & 8d & & \\
0 & d & 3d & 6d & & & & 6d & 11d & 17d & 24d & & & \\
0 & d & 4d & & & & & 4d & 15d & 32d & & & & \\
0 & d & & & & & & d & 16d & & & & & \\
0 & & & & & & & 0 & & & & & &
\end{array}
$$

This divides our proof of Long's idealized theorem into two subgoals:

1. Prove the generalization of Moessner's idealized theorem seeded with a constant d instead of a 1, which we call Long's weak theorem, and

2. prove the correctness of the decomposition of a composite sieve into two separate sieves, from which we can prove Long's idealized theorem.

Thus, it is clear that our next task is to prove Long's weak theorem.

10.2 Proving Long's weak theorem

As pointed out in the previous section, we are able to reduce Long's theorem into two separate sieves, each parameterized over a constant, c or d, that follows the exact same pattern as the dual sieve of Moessner's idealized theorem, as seen by the following figures,

	0	0	0	0	0	0
1	1	1	1	1	1	
0	1	2	3	4		
0	1	3	6			
0	1	4				
0	1					
0						

	0	0	0	0	0	0
1	1	1	1	1	1	
4	5	6	7	8		
6	11	17	24			
4	15	32				
1	16					
0						

	0	0	0	0	0	0
$1d$	$1d$	$1d$	$1d$	$1d$	$1d$	
0	$1d$	$2d$	$3d$	$4d$		
0	$1d$	$3d$	$6d$			
0	$1d$	$4d$				
0	$1d$					
0						

	0	0	0	0	0	0
$1d$	$1d$	$1d$	$1d$	$1d$	$1d$	
$4d$	$5d$	$6d$	$7d$	$8d$		
$6d$	$11d$	$17d$	$24d$			
$4d$	$15d$	$32d$				
$1d$	$16d$					
0						

where the only difference between the two sieves is the addition of the constant d to all entries in the second sieve. Since the structure of the two sieves are completely identical, all we have to do in order to prove Long's weak theorem is to add the parameter d to all our existing definitions and proofs, which we used to prove Moessner's idealized theorem.

Hence, we start by stating Long's weak theorem as,

```
Theorem Long_s_weak_theorem :
  ∀ (b e d : nat),
    p_moessner_stream b e d ∼ d ⊗ successive_powers b e.
```

where p_moessner_stream is identical to moessner_stream except that it is also parameterized over the constant d,

```
CoFixpoint p_moessner_stream (n r d : nat) : Stream nat :=
  (nth 0 (hypotenuse
           (nth n
                (create_triangles_vertically
                   n
                   (tuple_constant (S (S r)) 0)
                   (Str_prefix (S (S r)) (p_monomials 0 r 0 d)))
                [])) 1)
    ::: (p_moessner_stream (S n) r d).
```

which is propagated to the seed tuple expressed in terms of p_monomials,

```
CoFixpoint p_monomials (t r n d : nat) : Stream nat :=
  (p_monomial t r n d) ::: (p_monomials t r (S n) d).
```

which again is identical to monomials except for the constant d. Likewise, it relies on p_monomial,

```
Definition p_monomial (x n k d : nat) : nat :=
  d * C(n,k) * x ^ k.
```

which multiplies a monomial of a binomial expansion with the constant d. The proof of Long_s_weak_theorem essentially follows from the two proofs,

```
Corollary hypotenuse_create_triangle_vertically_p_monomials :
  ∀ (r t d : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (p_monomials t r 0 d))) =
    (rev (Str_prefix (S r) (p_monomials (S t) r 0 d))).
```

and

```
Theorem nth_triangle_create_triangles_vertically_p_monomials :
  ∀ (n r t d : nat),
    nth n
      (create_triangles_vertically n
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (p_monomials t r 0 d)))
      [] =
    (create_triangle_vertically
       (tuple_constant (S (S r)) 0)
       (Str_prefix (S (S r)) (p_monomials (n + t) r 0 d))).
```

which are proved using the exact same proof script as in the case of Moessner's idealized theorem, except for the addition of the constant *d*, which does not affect any of the proofs. Thus, we skip going through the rest of the proofs of Long_s_weak_theorem, since we would simply be repeating what we have already said and proved in the previous chapters of this dissertation.

Having proved Long's weak theorem, we take the next logical step and prove Long's idealized theorem.

10.3 Proving Long's idealized theorem

Just as Long's weak theorem naturally generalizes Moessner's idealized theorem from a seed value of 1 to a constant d, we now generalize Long's weak theorem from a seed tuple with one nonzero entry, d, to two, c and d, from which we obtain Long's idealized theorem.

As always, we start out our proof from an example in order to build our intuition, thus we return to the sieve example from Section 10.1,

$$
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & \\
d & d & d & d & d & d & & d & d & d & d & d & \\
c & c+d & c+2d & c+3d & c+4d & & & c+5d & c+6d & c+7d & c+8d & & \\
0 & c+d & 2c+3d & 3c+6d & & & & 4c+11d & 5c+17d & 6c+24d & & & \\
0 & c+d & 3c+4d & & & & & 7c+15d & 12c+32d & & & & \\
0 & c+d & & & & & & 8c+16d & & & & & \\
0 & & & & & & & & & & & &
\end{array}
$$

about which we remember that we can view the sieve as the composition of two simpler sieves, one creating Moessner triangles of rank 3 containing cs and one creating Moessner triangles of rank 4 containing ds. Now, in order to prove this decomposition we start by proving a more general theorem,

```
Theorem hypotenuse_create_triangle_vertically_list_sum :
  ∀ (r : nat) (σ  τ : Stream nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant r 0)
         ((Str_prefix r σ ) ⊕
          (Str_prefix r τ ))) =
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant r 0)
          (Str_prefix r σ ))) ⊕
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant r 0)
          (Str_prefix r τ ))).
```

which states that taking the hypotenuse of a Moessner triangle created from a vertical seed tuple that is a sum of two tuples, here defined as the prefixes of two streams, (`Str_Prefix r σ`) and (`Str_Prefix r τ`), yields the same result as summing the hypotenuses of the Moessner triangles created from each of the two tuples. The decomposition theorem is proved by induction on the prefix length `r` and rewriting with existing equivalences, along with two new helper lemmas for dealing with the list heads in the inductive case,

```
Lemma length_of_list_sum :
  ∀ (xs ys : list nat),
    length (xs ⊕ ys) = max (length xs) (length ys).
```

and

```
Lemma nth_make_tuple_list_sum :
  ∀ (n r i j : nat) (σ  τ : Stream nat),
    nth n (make_tuple (Str_prefix r σ ) i) 0 +
    nth n (make_tuple (Str_prefix r τ ) j) 0 =
    nth n ((make_tuple (Str_prefix r σ ) i) ⊕
           (make_tuple (Str_prefix r τ ) j)) 0.
```

We prove `length_of_list_sum` by structural induction on the first list, `xs`, and case analysis on the second list, `ys`, while we prove `nth_make_tuple_-list_sum` by induction on the element index, `n`, and case analysis on the prefix length, `r`. This completes the proof of `hypotenuse_create_triangle_-vertically_list_sum`. As a corollary of the above theorem, we obtain,

```
Corollary hypotenuse_create_triangle_vertically_p_monomials_list_sum :
  ∀ (r t d c : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S (S r))) 0)
         ((Str_prefix (S (S (S r))) (0 ::: (p_monomials t r 0 c))) ⊕
          (Str_prefix (S (S (S r))) (p_monomials t (S r) 0 d)))) =
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant (S (S (S r))) 0)
          (Str_prefix (S (S (S r))) (0 ::: (p_monomials t r 0 c))))) ⊕
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant (S (S (S r))) 0)
          (Str_prefix (S (S (S r))) (p_monomials t (S r) 0 d)))).
```

by simply instantiating σ and τ to `(0 ::: (p_monomials t r 0 c))` and `(p_monomials t (S r) 0 d)`, respectively, which describes the situation in our initial example for any rank, `r`, and constants, `c` and `d`, but only for one triangle. Hence, if we let `t = 0` and `r = 3` we get the decomposition of our example sieve for the first Moessner triangle.

Having proved that we can decompose a seed tuple expressed as a sum of two tuples filled with monomials parameterized over cs and ds, we move on to prove that the nth Moessner triangle, created from an initial configuration consisting of a seed tuple expressed in terms of a sum, can also be described in terms of a seed tuple over a similar sum,

```
Theorem nth_triangle_create_triangles_vertically_p_monomials_list_sum :
  ∀ (n r t d c : nat),
    nth n
      (create_triangles_vertically n
         (tuple_constant (S (S (S r))) 0)
         ((Str_prefix (S (S (S r))) (0 ::: (p_monomials t r 0 c))) ⊕
          (Str_prefix (S (S (S r))) (p_monomials t (S r) 0 d)))) [] =
    create_triangle_vertically
      (tuple_constant (S (S (S r))) 0)
      ((Str_prefix (S (S (S r))) (0 ::: (p_monomials (n + t) r 0 c))) ⊕
       (Str_prefix (S (S (S r))) (p_monomials (n + t) (S r) 0 d))).
```

We prove the above theorem by induction on the triangle index, `n`, and using the just proved `hypotenuse_create_triangle_vertically_p_monomials_-list_sum` along with,

```
Corollary hypotenuse_create_triangle_vertically_p_monomials :
  ∀ (r t d : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (p_monomials t r 0 d))) =
    (rev (Str_prefix (S r) (p_monomials (S t) r 0 d))).
```

which we proved as part of Long's weak theorem. Furthermore, we also need a few extra helper lemmas to deal with the padding of the stream prefixes and tuples,

```
Lemma hypotenuse_create_triangle_vertically_remove_padding :
  ∀ (r : nat) (σ : Stream nat),
    hypotenuse (create_triangle_vertically
                 (tuple_constant (S r) 0)
                 (Str_prefix (S r) (0 ::: σ))) =
    hypotenuse (create_triangle_vertically
                 (tuple_constant (S r) 0)
                 (Str_prefix r σ)).
```

and

```
Lemma create_triangle_vertically_horizontal_seed_tuple_padding :
  ∀ (r : nat) (σ : Stream nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S r) σ)) =
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant (S r) 0)
          (Str_prefix (S r) σ))) ++ [0].
```

both of which are proved by induction on the stream prefix length, `r`, thus proving `nth_triangle_create_triangles_vertically_p_monomials_-list_sum`.

Now, by combining the theorem `hypotenuse_create_triangle_-vertically_p_monomials_list_sum` with the theorem `nth_triangle_-create_triangles_vertically_p_monomials_list_sum` we can prove the following corollary,

```
Corollary bottom_element_of_nth_triangle_is_power_p_monomials_list_sum :
  ∀ (n r d c : nat),
    nth 0
        (hypotenuse
           (nth n
               (create_triangles_vertically
                  n
                  (tuple_constant (S (S (S r))) 0)
                  ((Str_prefix (S (S (S r)))
                              (0 ::: (p_monomials 0 r 0 c))) ⊕
                   (Str_prefix (S (S (S r)))
                              (p_monomials 0 (S r) 0 d)))) [])) 1 =
    (c * (S n) ^ r) + (d * (S n) ^ (S r)).
```

which states the desired sum of monomials,

```
(c * (S n) ^ r) + (d * (S n) ^ (S r)),
```

originally stated in Formula 10.1 and adapted to our Coq formalization, that we want as the result sequence of Long's idealized theorem. Hence, we now repeat the steps we used to prove Moessner's idealized theorem, by first defining a `long_stream`,

```
CoFixpoint long_stream (n r d c : nat) : Stream nat :=
  (nth 0 (hypotenuse
           (nth n
               (create_triangles_vertically
                  n
                  (tuple_constant (S (S (S r))) 0)
                  ((Str_prefix (S (S (S r)))
                              (0 ::: (p_monomials 0 r 0 c))) ⊕
                  (Str_prefix (S (S (S r)))
                              (p_monomials 0 (S r) 0 d)))) [])) 1)
    ::: (long_stream (S n) r d c).
```

which enumerates the bottom values of the successive Moessner triangles created by the composite dual sieve of Long's idealized theorem, that we then use to state Long's idealized theorem,

```
Theorem Long_s_theorem :
  ∀ (b e d c : nat),
    long_stream b e d c ∼
    (c ⊗ (successive_powers b e)) ⊕ (d ⊗ (successive_powers b (S e))).
```

which we prove by coinduction and by rewriting the initial value with `bottom_element_of_nth_triangle_is_power_p_monomials_list_sum`.

So far, we have generalized Moessner's idealized theorem from a seed value of 1 to a constant d, giving us Long's weak theorem, which we further generalized by extending the initial configuration from a seed tuple of one nonzero entry, d, to two nonzero entries, c and d, thus proving Long's idealized theorem. Consequently, we investigate the possibility of taking the generalization one step further by going from a seed tuple of two nonzero entries to a seed tuple with an arbitrary number of nonzero entries.

10.4 Beyond Long's theorem

Since Long's idealized theorem describes the result sequence generated by Moessner's sieve, when starting from a seed tuple of two constants, c and d,

$$
\begin{array}{c:r:r:r:r:r:r:r:r:r:r:r:r}
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & & & \\
d & d & d & d & d & d & & d & d & d & d & d & \\
c & c+d & c+2d & c+3d & c+4d & & & c+5d & c+6d & c+7d & c+8d & & \\
0 & c+d & 2c+3d & 3c+6d & & & & 4c+11d & 5c+17d & 6c+24d & & & \\
0 & c+d & 3c+4d & & & & & 7c+15d & 12c+32d & & & & \\
0 & c+d & & & & & & 8c+16d & & & & & \\
0 & & & & & & & & & & & &
\end{array}
$$

we now ask the obvious question of what happens if we start from a seed tuple of 3 or even n values? Looking at the result sequence of the above sieve, we know that it enumerates the values of the binomial, $c \cdot (1+t)^3 + d \cdot (1+t)^4$, which gives us the idea to label $c = a_3$ and $d = a_4$, and fill the rest of the seed tuple with a_i,

$$
\begin{array}{r:r:r:r:r:r:r}
 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & \\
a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & \\
a_3 & a_3+a_4 & a_3+2a_4 & a_3+3a_4 & a_3+4a_4 & & \\
a_2 & a_2+a_3+a_4 & a_2+2a_3+3a_4 & a_2+3a_3+6a_4 & & & \\
a_1 & a_1+a_2+a_3+a_4 & a_1+2a_2+3a_3+4a_4 & & & & \\
a_0 & a_0+a_1+a_2+a_3+a_4 & & & & & \\
0 & & & & & &
\end{array}
$$

yielding the above Moessner triangle. Now, if we examine the entries of the hypotenuse in this triangle,

$$
\begin{array}{rcrcrcrcr}
 & & & & & & & & a_4 \\
 & & & & & & a_3 & + & 4a_4 \\
 & & & & a_2 & + & 3a_3 & + & 6a_4 \\
 & & a_1 & + & 2a_2 & + & 3a_3 & + & 4a_4 \\
a_0 & + & a_1 & + & a_2 & + & a_3 & + & a_4
\end{array}
$$

we notice that we can rearrange them into the following Pascal-like triangle,

$$\begin{array}{ccccccccc} &&&& a_0 &&&& \\ &&& a_1 && a_1 &&& \\ && a_2 && 2a_2 && a_2 && \\ & a_3 && 3a_3 && 3a_3 && a_3 & \\ a_4 && 4a_4 && 6a_4 && 4a_4 && a_4 \end{array}$$

where the sum of the entries yields the following result,

$$a_0 + 2a_1 + 4a_2 + 8a_3 + 16a_4,$$

located at the bottom of the first column of the second triangle of the sieve, which we can restate as,

$$a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + a_3 \cdot 2^3 + a_4 \cdot 2^4.$$

Likewise, if we calculated the next triangle and the subsequent first column, we would obtain the values,

$$a_0 + 3a_1 + 9a_2 + 27a_3 + 81a_4,$$

which we can once again restate as,

$$a_0 \cdot 3^0 + a_1 \cdot 3^1 + a_2 \cdot 3^2 + a_3 \cdot 3^3 + a_4 \cdot 3^4.$$

This observation suggests that the repeated application of `create_-triangle_vertically` on the vertical seed tuple,

$$a_4, a_3, a_2, a_1, a_0,$$

yields a result sequence that enumerates the values of the polynomial,

$$p(t) = \sum_{i=0}^{4} a_i \cdot (1+t)^i,$$

where t is the triangle index. Thus, we conjecture that applying the dual sieve on an initial configuration where the vertical seed tuple consists of the constants,

$$a_n, a_{n-1}, \ldots, a_1, a_0,$$

yields a sequence of Moessner triangles where the bottom elements, comprising the result sequence, enumerate the values of the polynomial,

$$p(t) = \sum_{i=0}^{n} a_i \cdot (1+t)^i.$$

While we do not have a formal proof for this conjecture yet, we have seen strong indications of its correctness throughout this chapter, as we can decompose a seed tuple consisting of any sum of two tuples and have proved that the conjecture holds for the binomial $a_{i+1} \cdot (1+t)^{i+1} + a_i \cdot (1+t)^i$, as stated by Long's idealized theorem.

The relation between Moessner's sieve and polynomial evaluation is further explored in the next chapter, where we relate Horner's method to Moessner's sieve.

10.5 Summary

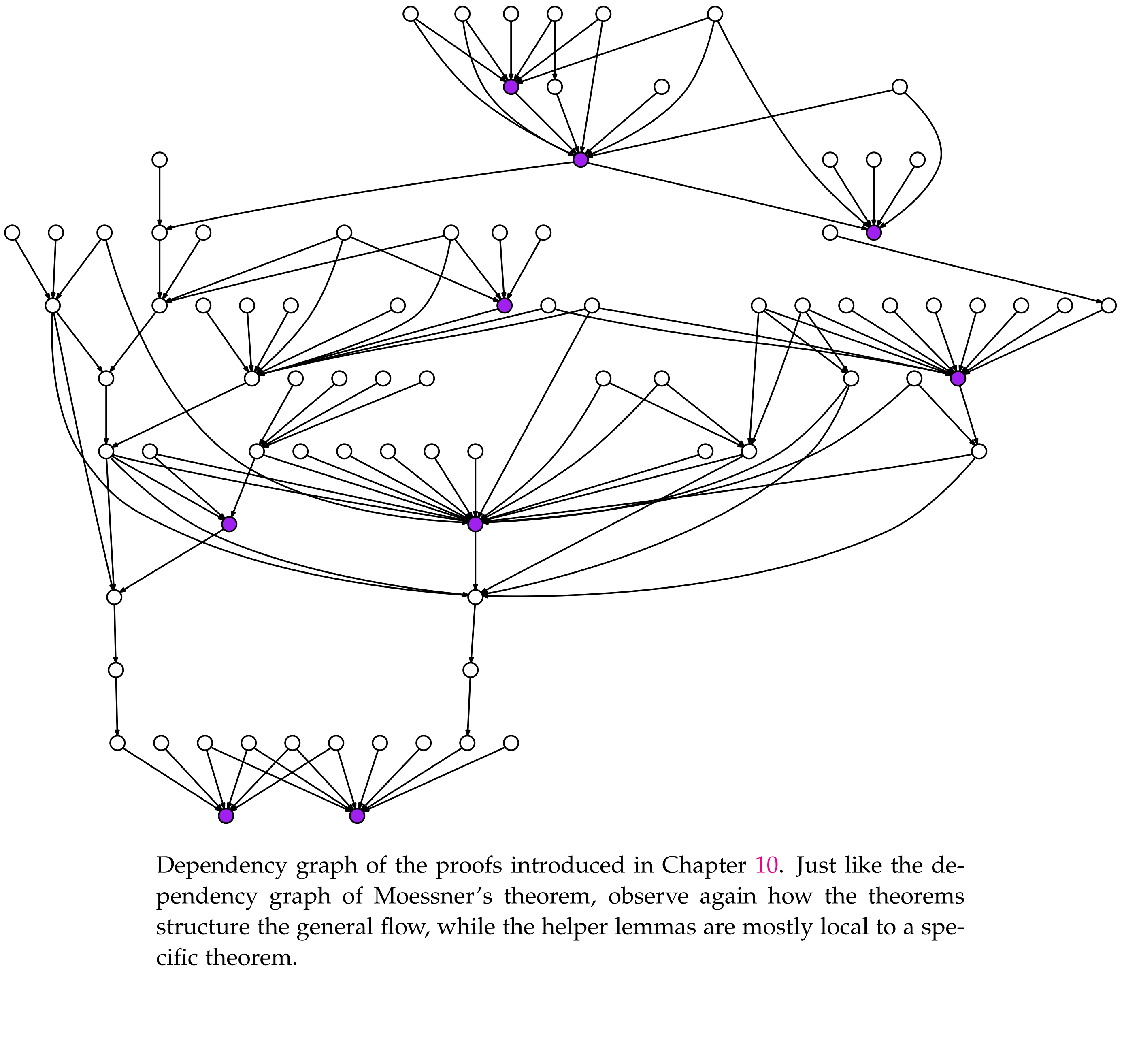
In this chapter, we have proved an idealized version of Long's theorem stated in terms of the dual of Moessner's sieve. Furthermore, we have conjectured a new generalization of Long's theorem that connects it to polynomial evaluation.

In order to state and prove Long's idealized theorem, we started by adapting Long's original theorem to our dual sieve which resulted in a series of generalizations leading to the division of Long's idealized theorem into two subgoals:

1. The generalization of Moessner's idealized theorem from a seed value of 1 to a constant d, which we have named Long's weak theorem, and

2. the generalization from one constant, d, to a pair of constants, c and d, which we have named Long's idealized theorem.

Proving the first part simply required the addition of the constant d in all definitions and proofs used for proving Moessner's idealized theorem, without having to change the proof script beyond that. The second part followed by showing that a seed tuple of any sum of two tuples can be decomposed into the sum of two sieves; one for each tuple. This observation lead to the conjecture of a generalization of Long's idealized theorem stated in terms of a tuple of constants.



Dependency graph of the proofs introduced in Chapter 10. Just like the dependency graph of Moessner's theorem, observe again how the theorems structure the general flow, while the helper lemmas are mostly local to a specific theorem.

Chapter 11

Deriving Moessner's sieve from Horner's method

"What makes the desert beautiful,"
said the little prince,
"is that somewhere it hides a well..."

ANTOINE DE SAINT-EXUPÉRY,
THE LITTLE PRINCE

The goal of this chapter is to derive Moessner's sieve from the procedure known as Horner's method. By doing so, we strengthen the connection between Moessner's sieve and polynomial evaluation, as conjectured in the previous chapter, while also expanding the space of possible applications of Moessner's sieve significantly, as Horner's method has several applications in both algebra [33] and combinatorics [13].

The chapter is structured as follows. In Section 11.1, we introduce the definition of Horner's method for polynomial evaluation and polynomial division, while also proving an equivalence relation between the two. Having covered the basics of Horner's method, we show how to obtain Taylor polynomials by repeated application of Horner's method in Section 11.2. In Section 11.3, we use the knowledge we have acquired in the previous two sections to derive Moessner's sieve from Horner's method.

11.1 Defining Horner's method

Horner's method [6, 16], named after the British mathematician William George Horner, refers to two procedures for evaluating or dividing a polynomial by means of the same recursive substitution scheme,[1] using the small-

[1] History has it that similar procedures to Horner's method had previously been used by a range of Chinese mathematicians between 100 BC and 1303, while the latest example preceding Horner is Paolo Ruffini in 1804, 15 years before Horner published his article before the Royal Society in 1819 [33].

est possible number of operations [26, 45].[2] The latter method for dividing a polynomial is often used together with Newton's method for finding the roots of a polynomial, but this falls out of the scope of this dissertation.

In this section, we first introduce Horner's method for polynomial evaluation followed by Horner's method for polynomial division, both of which we formalize in Coq and prove to have an equivalence relation.

11.1.1 Polynomial evaluation using Horner's method

In order to understand the advantages of using Horner's method for evaluating a polynomial, we first examine how this is usually done. If we let $p(x) = 7x^4 + 2x^3 + 5x^2 + 4x + 6$ and $x = 3$, then we would evaluate $p(3)$ one term at a time and sum all the intermediate results. However, by doing so we are unfortunately performing redundant operations when evaluating the exponents, as can be seen when we unfold the evaluation of the exponents,

$$\begin{aligned} p(3) &= 7 \cdot (3^4) + 2 \cdot (3^3) + 5 \cdot (3^2) + 4 \cdot (3) + 6 \\ &= 7 \cdot (3 \cdot 3 \cdot 3 \cdot 3) + 2 \cdot (3 \cdot 3 \cdot 3) + 5 \cdot (3 \cdot 3) + 4 \cdot (3) + 6. \end{aligned}$$

Here, the evaluation of the largest exponent, $3^4 = (3 \cdot 3 \cdot 3 \cdot 3)$, also calculates all exponents of a lesser degree, i.e., $3^3 = (3 \cdot 3 \cdot 3)$ and $3^2 = (3 \cdot 3)$, as its intermediate results. Luckily, we can transform the formula of the polynomial p in such a way that the operations calculating the exponents are shared across the terms. In fact, since the number of multiplications by 3 decreases by 1 for each term, we can nest the multiplications across the terms like so,

$$\begin{aligned} p(3) &= 7 \cdot (3 \cdot 3 \cdot 3 \cdot 3) + 2 \cdot (3 \cdot 3 \cdot 3) + 5 \cdot (3 \cdot 3) + 4 \cdot (3) + 6 \\ p(3) &= (7 \cdot (3 \cdot 3 \cdot 3) + 2 \cdot (3 \cdot 3) + 5 \cdot (3) + 4) \cdot 3 + 6 \\ p(3) &= ((7 \cdot (3 \cdot 3) + 2 \cdot (3) + 5) \cdot 3 + 4) \cdot 3 + 6 \\ p(3) &= (((7 \cdot 3 + 2) \cdot 3 + 5) \cdot 3 + 4) \cdot 3 + 6, \end{aligned} \tag{11.1}$$

thus removing any redundant multiplications used for evaluating the exponents. Formula 11.1 now exhibits a simple inductive structure which adds one multiplication and one addition for each term in the polynomial p. As a result, we can now evaluate the final formula of p,

$$p(3) = (((7 \cdot 3 + 2) \cdot 3 + 5) \cdot 3 + 4) \cdot 3 + 6, \tag{11.2}$$

by repeatedly performing a multiplication and an addition, starting from the innermost set of parentheses,

$$\begin{aligned} p(3) &= (((7 \cdot 3 + 2) \cdot 3 + 5) \cdot 3 + 4) \cdot 3 + 6 \\ &= ((23 \cdot 3 + 5) \cdot 3 + 4) \cdot 3 + 6 \\ &= (74 \cdot 3 + 4) \cdot 3 + 6 \\ &= 226 \cdot 3 + 6 \\ &= 684. \end{aligned} \tag{11.3}$$

[2]Operations refer strictly to addition and multiplication in this context.

If we compare the number of operations performed in the first and last equation of Formula 11.1, we count a total of 14 in the former and 8 in the latter. The difference of 6 operations corresponds exactly to the number of multiplications required to evaluate the exponents in the first equation. Thus, our transformation of the polynomial formula into its inductive form, has removed the computational overhead of evaluating each of the exponents in sequence. Lastly, it has even been proved that the number of additions and multiplications used in this procedure, are indeed the smallest number possible for evaluating a polynomial [26,45].

Upon closer examination of the intermediate results of Formula 11.3, we can make out a recursive substitution scheme happening under the hood,

$$
\begin{aligned}
7 &= 7 \\
23 &= 7 \cdot 3 + 2 \\
74 &= 23 \cdot 3 + 5 \\
226 &= 74 \cdot 3 + 4 \\
684 &= 226 \cdot 3 + 6.
\end{aligned} \tag{11.4}
$$

where each intermediate result is the result of multiplying the previous result by 3 and adding the next coefficient. If we assign the intermediate values on the left-hand side, $(7, 23, 74, 226, 684)$, to the variable b_i, assign the value 3 to the variable k, and lastly assign the values corresponding to the coefficients of p, $(7, 2, 5, 4, 6)$, to the variable a_i, we can restate Formula 11.4 like so,

$$
\begin{aligned}
b_4 &= a_4 \\
b_3 &= b_4 \cdot k + a_3 \\
b_2 &= b_3 \cdot k + a_2 \\
b_1 &= b_2 \cdot k + a_1 \\
b_0 &= b_1 \cdot k + a_0.
\end{aligned} \tag{11.5}
$$

Formula 11.5 now reflects a recursively structured, and easily generalizable, substitution procedure where $b_4 = a_4$ is the base case, and the inductive case is defined in terms of the next coefficient in the polynomial and the preceding intermediate result, $b_3 = b_4 \cdot k + a_3$. The procedure terminates when it reaches the last term of the polynomial p, where $b_0 = b_1 \cdot k + a_0$ is the result of evaluating $p(k)$.

We call the above procedure Horner's method [16] for polynomial evaluation, and formalize it in Coq by first representing a polynomial as a list of natural numbers,

```
Notation polynomial := (list nat).
```

for which we define the procedure,

```
Fixpoint horner_poly_eval_acc (cs : polynomial) (x a : nat) : nat :=
  match cs with
    | [] ⇒ a
    | c :: cs' ⇒ horner_poly_eval_acc cs' x (c + x * a)
  end.
```

which takes a `polynomial`, corresponding to a_i, an `x`, corresponding to k, and an accumulator, a, corresponding to the intermediate result b_i. As described above, it returns the final value of the accumulator, `a`, in the base case, and multiplies `a` by `x` for each recursive call and adds the coefficient `c`. Lastly, we define a wrapper procedure,

```
Definition horner_poly_eval (cs : polynomial) (x : nat) : nat :=
  horner_poly_eval_acc cs x 0.
```

which initializes the accumulator to `0`. As a result, we can now evaluate the example polynomial of Formula 11.2, $p(x) = 7x^4 + 2x^3 + 5x^2 + 4x + 6$, for $x = 3$, by passing the coefficients of p as the list `[7; 2; 5; 4; 6]`, along with `3` as the value of x, to `horner_poly_eval` like so,

```
horner_poly_eval [7; 2; 5; 4; 6] 3 = 684,
```

giving the expected result, `684`.

Having formalized Horner's method for polynomial evaluation, as the procedures `horner_poly_eval_acc` and `horner_poly_eval`, we now define Horner's method for polynomial division.

11.1.2 Polynomial division using Horner's method

Now that we have used Horner's method as an efficient procedure for evaluating a polynomial, using a recursive substitution scheme, we move on to examine its use for polynomial division.

According to the definition of polynomial division, when dividing two polynomials, p and d, $\frac{p(x)}{d(x)}$, where $d \neq 0$, the result is a quotient, q, and a remainder, r, satisfying the relation,

$$p(x) = d(x) \cdot q(x) + r(x), \tag{11.6}$$

where r has a degree less than d. For the goal of this chapter, we restrict ourselves to division with a binomial, $x - k$, which means that r is always a constant, and 0 when d divides p.

One procedure for polynomial division is polynomial long division, described in further details in the glossary, which we can use to divide the polynomial $p(x) = 2x^3 + 4x^2 + 11x + 3$ with the binomial $d(x) = x - 2$, giving us the following result,

$$\begin{array}{rl}
 & \phantom{2x^3 + {}} 2x^2 \quad + 8x + 27 \\ \cline{2-2}
x - 2 \big) & 2x^3 + 4x^2 + 11x \quad + 3 \\
 & \underline{-\,2x^3 + 4x^2} \\
 & \phantom{2x^3 + {}} 8x^2 + 11x \\
 & \phantom{2x^3 + {}} \underline{-\,8x^2 + 16x} \\
 & \phantom{2x^3 + 4x^2 + {}} 27x \quad + 3 \\
 & \phantom{2x^3 + 4x^2 + {}} \underline{-\,27x + 54} \\
 & \phantom{2x^3 + 4x^2 + 11x + {}} 57
\end{array}$$

where we can read the quotient, $2x^2 + 8x + 27$, from the line above the numerator, $2x^3 + 4x^2 + 11x + 3$, and we can read the remainder, 57, from the value below the last horizontal line in the calculation. Lastly, we can verify the calculations by checking that the relation in Formula 11.6 is satisfied,

$$2x^3 + 4x^2 + 11x + 3 = (x - 2)(2x^2 + 8x + 27) + 57.$$

If we examine the intermediate results of the procedure, $(2, 8, 27, 57)$, i.e., the leftmost values under each horizontal line, we can make out a similar recursive substitution scheme to what we saw in the case of polynomial evaluation,

$$\begin{aligned}
2 &= 2 \\
8 &= 2 \cdot 2 + 4 \\
27 &= 8 \cdot 2 + 11 \\
57 &= 27 \cdot 2 + 3.
\end{aligned}$$

where each intermediate result is equal to the previous result multiplied by the second term of the denominator, $x - 2$, plus the next coefficient. This time we assign the intermediate results on the left to the variable b_{i-1}, the last result to the variable r, the second term of the denominator to the variable k, and the coefficients of p to the variable a_i, which yields the following set of equations,

$$\begin{aligned}
b_2 &= a_3 \\
b_1 &= b_2 \cdot k + a_2 \\
b_0 &= b_1 \cdot k + a_1 \\
r &= b_0 \cdot k + a_0.
\end{aligned}$$

These equations strongly suggest that we can divide p with d using the same recursive substitution procedure, as described in the evaluation case, spending just one addition and multiplication per term, which again reduces the number of operations to a minimum. Furthermore, we can put the substitution scheme above in a tabular format, similar to polynomial long division,

$$\begin{array}{c|cccc} & a_3 & a_2 & a_1 & a_0 \\ k & & b_2 \cdot k & b_1 \cdot k & b_0 \cdot k \\ \hline & a_3 & b_2 \cdot k + a_2 & b_1 \cdot k + a_1 & b_0 \cdot k + a_0 \\ & = b_2 & = b_1 & = b_0 & = r \end{array} \tag{11.7}$$

where the coefficients of the polynomial are located at the top row, the second term of the denominator to the far left, and the coefficients of the resulting quotient, b_2, b_1, b_0, and the remainder, r, at the bottom row of the table.

We formalize the tabular representation in Formula 11.7 as the following procedure,

```
Fixpoint horner_poly_div_acc (cs' : polynomial) (x a : nat) :
  polynomial :=
  match cs' with
    | [] ⇒ []
    | c' :: cs'' ⇒
      (c' + (x * a)) :: (horner_poly_div_acc cs'' x (c' + (x * a)))
  end.
```

which performs the exact same substitution scheme as in horner_poly_-eval_acc, except that it also aggregates the intermediate results and adds them to the result polynomial. Likewise, we define a wrapper function,

```
Definition horner_poly_div (cs : polynomial) (x : nat) : polynomial :=
  match cs with
    | [] ⇒ []
    | c :: cs' ⇒ c :: (horner_poly_div_acc cs' x c)
  end.
```

which sets the initial accumulator to the first coefficient and adds it to the result polynomial. Now, if we wanted to divide our initial polynomial $p(x) = 2x^3 + 4x^2 + 11x + 3$ with the binomial $d(x) = x - 2$, we would pass the list [2; 4; 11; 3] as the input polynomial cs and 2 as the input value x to horner_poly_div, from which we would get the result list [2; 8; 27; 57], where [2; 8; 27] are the coefficient of the quotient and 57 is the remainder. Thus, we have now defined Horner's method for polynomial division as the procedures horner_poly_div_acc and horner_poly_div.

Next, we prove an equivalence relation between the two procedures for polynomial evaluation and polynomial division using Horner's method.

11.1.3 Equivalence of the two Horner procedures

Due to the strong similarity between the procedure for polynomial evaluation and the procedure for polynomial division, we are interested in stating an equivalence relation between the two. As such, we note that the last element in the result polynomial of horner_poly_div is equal to the result of horner_poly_eval when given the same input,

```
Theorem horner_poly_eval_eq_horner_poly_div :
  ∀ (cs : polynomial) (x : nat),
    horner_poly_eval cs x =
    last (horner_poly_div cs x) 0.
```

Proving the relation requires us to first prove a similar equivalence relation between the underlying procedures `horner_poly_div_acc` and `horner_poly_eval_acc`, parameterized over the accumulator, `a`,

```
Theorem horner_poly_eval_acc_eq_horner_poly_div_acc :
  ∀ (cs' : polynomial) (c x a : nat),
    horner_poly_eval_acc (c :: cs') x a =
    last (horner_poly_div_acc cs' x (c + x * a)) (c + x * a).
```

We prove the underlying theorem by structural induction on the polynomial, `cs`, allowing us to prove the original equivalence by case analysis on the polynomial, `cs`. Incidentally, the theorem `horner_poly_eval_eq_horner_poly_div` also proves an implementation-specific version of the polynomial remainder theorem, which we state in the next section.

Having motivated and formalized Horner's method for polynomial evaluation and polynomial division, as the two procedures `horner_poly_eval` and `horner_poly_div`, and proved their equivalence, we proceed by describing how to obtain Taylor polynomials using Horner's method.

11.2 Obtaining Taylor Polynomials

Having covered the definition of Horner's method for polynomial evaluation and polynomial division, we show how to calculate Taylor polynomials using Horner's method.

In the first section we state two preliminary theorems, the polynomial remainder theorem and Taylor's theorem, which we then use to show how to generate Taylor polynomials using Horner's method.

11.2.1 Preliminaries

Below, we first state the polynomial remainder theorem followed by the definitions of Taylor series and Taylor polynomials, which we use to finally state Taylor's theorem.

As pointed out in the previous section, if we divide a polynomial, p, with a binomial, $x - k$, the remainder of the division is equal to $p(k)$, which is captured by the polynomial remainder theorem.

Theorem 2 (Polynomial remainder theorem)**.** *Given a polynomial,*

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

where $a_0, \ldots, a_n \in \mathbb{N}$, and a binomial,

$$d(x) = x - k,$$

where $k \in \mathbb{N}$, the remainder of dividing p with d, denoted r, is equal to $p(k)$. Furthermore, d divides p if and only if $p(k) = 0$.

Next, we define Taylor series and Taylor polynomials in order to state Taylor's theorem. A Taylor series is the representation of a function as an infinite sum of terms, calculated from the values of the function's derivatives at a specific point.

Definition 1 (Taylor series)**.** Given a function p and a natural number k, the Taylor series of p is,

$$\frac{p(k)}{0!}(x-k)^0 + \frac{p'(k)}{1!}(x-k)^1 + \frac{p''(k)}{2!}(x-k)^2 + \frac{p^{(3)}(k)}{3!}(x-k)^3 + \cdots,$$

which can be written as,

$$\sum_{i=0}^{\infty} \frac{p^{(i)}(k)}{i!}(x-k)^i.$$

A Taylor series with a finite number of terms, $n \in \mathbb{N}$, is called a Taylor polynomial and written,

$$\sum_{i=0}^{n} \frac{p^{(i)}(k)}{i!}(x-k)^i.$$

Since we are working solely with polynomials, we are able to restate any polynomial as a Taylor polynomial, calculating the exact same values. This brings us to the following simplified version of Taylor's theorem defined over polynomials and natural numbers.

Theorem 3 (Taylor's theorem)**.** *Given a polynomial, p, and two natural numbers, n and k, the n-th order Taylor polynomial of p, $P_{n,k}$, at the point k is,*

$$P_{n,k}(x) = \sum_{i=0}^{n} \frac{p^{(i)}(k)}{i!}(x-k)^i. \tag{11.8}$$

Proving Taylor's theorem falls out of the scope of this dissertation. Having stated the above definitions and theorems, we now show how to obtain Taylor polynomials using Horner's method.

11.2.2 Generating Taylor polynomials

From Theorem 3, we know that given a polynomial,

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

where $a_0, \ldots, a_n \in \mathbb{N}$, and a $k \in \mathbb{N}$, the Taylor polynomial of p at point k is,

$$P_{n,k}(x) = \sum_{i=0}^{n} \frac{p^{(i)}(k)}{i!}(x-k)^i,$$

where every occurrence of the variable x has been substituted with $x - k$ and every coefficient a_i has been substituted with $\frac{p^{(i)}(k)}{i!}$. Thus, we need a way to compute these new values using Horner's method.

If we let $p(x) = 2x^3 + 4x^2 + 11x + 3$ and $k = 2$, we can calculate the coefficients of $P_{3,2}$ – without the use of Horner's method – by evaluating p and its first three derivatives for $x = 2$,

$$\frac{p(2)}{0!} = \frac{2 \cdot 2^3 + 4 \cdot 2^2 + 11 \cdot 2 + 3}{0!} = \frac{57}{0!} = 57 \tag{11.9}$$

$$\frac{p'(2)}{1!} = \frac{6 \cdot 2^2 + 8 \cdot 2 + 11}{1!} = \frac{51}{1!} = 51 \tag{11.10}$$

$$\frac{p''(2)}{2!} = \frac{12 \cdot 2 + 8}{2!} = \frac{32}{2!} = 16 \tag{11.11}$$

$$\frac{p^{(3)}(2)}{3!} = \frac{12}{3!} = 2, \tag{11.12}$$

which yields the 3-rd order Taylor polynomial of p at point 2,

$$\begin{aligned} P_{3,2}(x) &= \frac{p(2)}{0!}(x-2)^0 + \frac{p'(2)}{1!}(x-2)^1 \\ &\quad + \frac{p''(2)}{2!}(x-2)^2 + \frac{p^{(3)}(2)}{3!}(x-2)^3 \\ P_{3,2}(x) &= 57(x-2)^0 + 51(x-2)^1 + 16(x-2)^2 + 2(x-2)^3 \\ P_{3,2}(x) &= 2(x-2)^3 + 16(x-2)^2 + 51(x-2) + 57. \end{aligned} \tag{11.13}$$

Looking at the calculations above, we do not only have to evaluate four polynomials and divide each of them with a factorial, but we also have to take the repeated derivative of p. It would be useful if we could calculate these values using our existing definitions. From Theorem 2, we know that dividing p with a binomial, $d(x) = x - 2$,

	x^3	x^2	x^1	x^0
	2	4	11	3
2		4	16	54
	2	8	27	57

yields the quotient $q_0(x) = 2x^2 + 8x + 27$ and remainder $r_0 = p(2)$, which is also equal to $\frac{p(2)}{0!}$, since $0! = 1$. This corresponds to the result of Formula 11.9, which is also why we have subscripted the remainder with a 0, since it is the value of the coefficient of $P_{3,2}$ with index $i = 0$,

$$r_0 = \frac{p(2)}{0!} = 57.$$

Furthermore, it turns out that if we keep dividing the obtained quotient, a pattern emerges that connects the remainders of the subsequent divisions with the remaining coefficients of $P_{3,2}$. If we divide the quotient of the first division, $q_0(x) = 2x^2 + 8x + 27$, with the same binomial as before, $d(x) = x - 2$,

$$\begin{array}{c|ccc} & x^2 & x^1 & x^0 \\ & 2 & 8 & 27 \\ 2 & & 4 & 24 \\ \hline & 2 & 12 & 51 \end{array}$$

we get the quotient $q_1(x) = 2x + 12$ and remainder $r_1 = 51$. In line with the previous result, we notice that the remainder, r_1, is equal to the result of Formula 11.10, i.e., the value of the coefficient of $P_{3,2}$ with index $i = 1$,

$$r_1 = \frac{p'(2)}{1!} = 51.$$

If we repeat this procedure once more with the quotient $q_1(x) = 2x + 12$,

$$\begin{array}{c|cc} & x^1 & x^0 \\ & 2 & 12 \\ 2 & & 4 \\ \hline & 2 & 16 \end{array}$$

we get the remainder $r_2 = 16$, which matches the coefficient with index $i = 2$ in Formula 11.11,

$$r_2 = \frac{p''(2)}{2!} = 16,$$

and the quotient $q_2 = 2$, which is also equal to the last remainder, r_3, since q_2 is constant, and therefore it is also equal to the coefficient with index $i = 3$ in Formula 11.12,

$$q_2 = r_3 = \frac{p^{(3)}(2)}{3!} = 2.$$

Now, with the following coefficients in hand,

$$\begin{aligned} r_3 &= \frac{p'''(2)}{3!} = 2 \\ r_2 &= \frac{p''(2)}{2!} = 16 \\ r_1 &= \frac{p'(2)}{1!} = 51 \\ r_0 &= \frac{p(2)}{0!} = 57, \end{aligned}$$

the 3-rd order Taylor polynomial of p at point 2 becomes,

$$\begin{aligned} P_{3,2}(x) &= \frac{p(2)}{0!}(x-2)^0 + \frac{p'(2)}{1!}(x-2)^1 + \frac{p''(2)}{2!}(x-2)^2 + \frac{p^{(3)}(2)}{3!}(x-2)^3 \\ P_{3,2}(x) &= r_0(x-2)^0 + r_1(x-2)^1 + r_2(x-2)^2 + r_3(x-2)^3 \\ P_{3,2}(x) &= 57(x-2)^0 + 51(x-2)^1 + 16(x-2)^2 + 2(x-2)^3 \\ P_{3,2}(x) &= 2(x-2)^3 + 16(x-2)^2 + 51(x-2) + 57, \end{aligned}$$

which is equal to the last Taylor polynomial in Formula 11.13. Thus, we have demonstrated how to obtain the Taylor polynomial of a polynomial p at a point k, by repeatedly dividing the resulting quotient polynomials with a binomial, $x - k$, using Horner's method, where p is the initial polynomial to be divided [33]. Lastly, we can write the repeated application of Horner's method in a tabular format,

$$\begin{array}{r|rrrr} 2 & 2 & 4 & 11 & 3 \\ & & 4 & 16 & 54 \\ & 2 & 8 & 27 & \mathbf{57} \\ & & & & \\ & & 4 & 24 & \\ & 2 & 12 & \mathbf{51} & \\ & & & & \\ & & 4 & & \\ & 2 & \mathbf{16} & & \\ & & & & \\ & \mathbf{2}. & & & \end{array} \tag{11.14}$$

where the divisions are merged into a triangular array, such that the hypotenuse of the triangle, highlighted in boldface, enumerates the coefficients of the resulting Taylor polynomial.

We call this construction a Horner block and formalize it by first defining a `block` to be a `list` of `polynomials`,

```
Notation block := (list polynomial).
```

allowing us to introduce the following procedure,

```
Fixpoint create_horner_block_acc (n x : nat) (cs : polynomial) : block :=
  match n with
    | O ⇒ []
    | S n' ⇒ let cs' := removelast (horner_poly_div cs x) in
             cs' :: create_horner_block_acc n' x cs'
  end.
```

which performs the repeated application of Horner's method for polynomial division, while removing the last entry of each intermediate results. Here, `cs` and `x` denote the same as in the case of `horner_poly_div`, while `n` specifies the number of divisions. However, we note that there exists an extra base case in Formula 11.14, as no value is dropped from the initial `polynomial` in the Horner block. Hence, we define the wrapper,

```
Definition create_horner_block (n x : nat) (cs : polynomial) : block :=
  match n with
    | 0 ⇒ []
    | S n' ⇒ let cs' := horner_poly_div cs x in
             cs' :: create_horner_block_acc n' x cs'
  end.
```

which performs a single division without removing the last entry, followed by a call to `create_horner_block_acc`. Thus, we can obtain the Taylor polynomial of a polynomial p at a point k by reading the `hypotenuse` of the `block` returned by `create_horner_block` when given a list of p's coefficients and a value of k.

11.3 Derivation of Moessner's sieve

Having covered polynomial division using Horner's method and how to generate Taylor polynomials, we now show how these can be used to emulate Moessner's sieve. In order to do so, we first have to discuss a few extra properties of Horner's method.

If we let $p(x) = x^3$ and want to obtain the Taylor polynomial $P_{3,3}$, we can do so in two ways:

1. We repeatedly divide p with $x - 3$ and obtain the hypotenuse corresponding to the coefficients of $P_{3,3}$,

$$\begin{array}{c|cccc}
 & x^3 & x^2 & x^1 & x^0 \\
3 & 1 & 0 & 0 & 0 \\
 & & 3 & 9 & 27 \\
 & 1 & 3 & 9 & \mathbf{27} \\
 & & & & \\
 & & 3 & 18 & \\
 & 1 & 6 & \mathbf{27} & \\
 & & & & \\
 & & 3 & & \\
 & 1 & \mathbf{9} & & \\
 & & & & \\
 & \mathbf{1} & & &
\end{array}$$

2. We repeatedly divide p with $x - 1$, obtain the Taylor polynomial $P_{3,1}$ which we again repeatedly divide with $x - 1$ to get $P_{3,2}$, and lastly repeatedly divide $P_{3,2}$ with $x - 1$ to get the coefficients of $P_{3,3}$,

$$\begin{array}{c|cccc c|cccc c|cccc}
 & x^3 & x^2 & x^1 & x^0 & & x^3 & x^2 & x^1 & x^0 & & x^3 & x^2 & x^1 & x^0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 3 & 3 & 1 & 1 & 1 & 6 & 12 & 8 \\
 & & 1 & 1 & 1 & & & 1 & 4 & 7 & & & 1 & 7 & 19 \\
 & 1 & 1 & 1 & 1 & & 1 & 4 & 7 & 8 & & 1 & 7 & 19 & \mathbf{27} \\
 & & & & & & & & & & & & & & \\
 & & 1 & 2 & & & & 1 & 5 & & & & 1 & 8 & \\
 & 1 & 2 & 3 & & & 1 & 5 & 12 & & & 1 & 8 & \mathbf{27} & \\
 & & & & & & & & & & & & & & \\
 & & 1 & & & & & 1 & & & & & 1 & & \\
 & 1 & 3 & & & & 1 & 6 & & & & 1 & \mathbf{9} & & \\
 & & & & & & & & & & & & & & \\
 & 1 & & & & & 1 & & & & & \mathbf{1} & & &
\end{array} \tag{11.15}$$

From the above calculations, we first observe that the two result hypotenuses – highlighted in boldface – are identical. Secondly, we observe that the first remainder calculated in each of the three triangles in Formula 11.15, $(1, 8, 27)$, are equal to the powers of 3, $(1^3, 2^3, 3^3)$, and thus equal to the values of $p(1)$, $p(2)$ and $p(3)$, which demonstrates that Horner's method can be used to enumerate the values of p for the set of positive natural numbers. Lastly, upon closer examination of the procedure used above, we note that given a polynomial,

$$p(x) = a_3x^3 + a_2x^2 + a_1x + a_0,$$

the repeated division of p with $x - 1$ has the following structure,

$$\begin{array}{cccc}
a_3 & a_2 & a_1 & a_0 \\
 & a_3 & a_3 + a_2 & a_3 + a_2 + a_1 \\
a_3 & a_3 + a_2 & a_3 + a_2 + a_1 & \mathbf{a_3 + a_2 + a_1 + a_0} \\
 & & & \\
 & a_3 & 2a_3 + a_2 & \\
a_3 & 2a_3 + a_2 & \mathbf{3a_3 + 2a_2 + a_1} & \\
 & & & \\
 & a_3 & & \\
a_3 & \mathbf{3a_3 + a_2} & & \\
 & & & \\
\mathbf{a_3} & & &
\end{array}$$

where every non-shifted row, starting with the first row,

$$\begin{array}{cccc}
a_3 & a_2 & a_1 & a_0 \\
a_3 & a_3 + a_2 & a_3 + a_2 + a_1 & \mathbf{a_3 + a_2 + a_1 + a_0} \\
a_3 & 2a_3 + a_2 & \mathbf{3a_3 + 2a_2 + a_1} & \\
a_3 & \mathbf{3a_3 + a_2} & & \\
\mathbf{a_3} & & &
\end{array} \tag{11.16}$$

is the partial sum of the former non-shifted row. If we take the results of Formula 11.15 and strip away the left-most column and the top row, containing the value of k and the exponents, we get the following three Horner blocks:

$$\begin{array}{cccccccccccccc}
1 & 0 & 0 & 0 & \quad & 1 & 3 & 3 & 1 & \quad & 1 & 6 & 12 & 8 \\
 & 1 & 1 & 1 & & & 1 & 4 & 7 & & & 1 & 7 & 19 \\
1 & 1 & 1 & \mathbf{1} & & 1 & 4 & 7 & \mathbf{8} & & 1 & 7 & 19 & \mathbf{27} \\
 & 1 & 2 & & & & 1 & 5 & & & & 1 & 8 & \\
1 & 2 & \mathbf{3} & & & 1 & 5 & \mathbf{12} & & & 1 & 8 & \mathbf{27} & \\
 & 1 & & & & & 1 & & & & & 1 & & \\
1 & \mathbf{3} & & & & 1 & \mathbf{6} & & & & 1 & \mathbf{9} & & \\
\mathbf{1} & & & & & \mathbf{1} & & & & & \mathbf{1} & & &
\end{array} \tag{11.17}$$

Here, we note the regular structure of the blocks where every block is created from the hypotenuse of the previous block. Next, we perform the same transformation on Formula 11.17 as seen in Formula 11.16, where every shifted row is removed in order to expose the partial summation pattern between each intermediate result:

$$\begin{array}{llll} 1&0&0&0\\ 1&1&1&\mathbf{1}\\ 1&2&\mathbf{3}&\\ 1&\mathbf{3}&&\\ \mathbf{1}&&& \end{array} \qquad \begin{array}{llll} 1&3&3&1\\ 1&4&7&\mathbf{8}\\ 1&5&\mathbf{12}&\\ 1&\mathbf{6}&&\\ \mathbf{1}&&& \end{array} \qquad \begin{array}{llll} 1&6&12&8\\ 1&7&19&\mathbf{27}\\ 1&8&\mathbf{27}&\\ 1&\mathbf{9}&&\\ \mathbf{1}&&& \end{array} \tag{11.18}$$

Lastly, we remove the redundant rows, which appear as both the hypotenuse of one block and the initial row of the subsequent block, e.g., $(1, 3, 3, 1)$ is both the hypotenuse of the first Horner block and also the first row of the second Horner block. Furthermore, we pile the blocks on top of each other,

$$\begin{array}{llll} 1&0&0&0\\ 1&1&1&\mathbf{1}\\ 1&2&\mathbf{3}&\\ 1&\mathbf{3}&&\\ \mathbf{1}&&&\\ 1&4&7&\mathbf{8}\\ 1&5&\mathbf{12}&\\ 1&\mathbf{6}&&\\ \mathbf{1}&&&\\ 1&7&19&\mathbf{27}\\ 1&8&\mathbf{27}&\\ 1&\mathbf{9}&&\\ \mathbf{1}&&& \end{array} \tag{11.19}$$

resulting in a rotated mirror image of Moessner's sieve,

$$\begin{array}{cccccccccccc} 1&1&1&\mathbf{1}&1&1&1&\mathbf{1}&1&1&1&\mathbf{1}\\ 1&2&\mathbf{3}&&4&5&\mathbf{6}&&7&8&\mathbf{9}&\\ 1&\mathbf{3}&&&7&\mathbf{12}&&&19&\mathbf{27}&&\\ \mathbf{1}&&&&\mathbf{8}&&&&\mathbf{27}&&& \end{array}$$

where the right-most column enumerates the successive powers of x^3, which is the statement of Moessner's theorem for $n = 3$ and the observation made by Van Yzeren [44]. Now, if we examine Formula 11.19 from the perspective of our dual sieve, we observe that the rows of the Horner-based sieve do indeed enumerate the columns of the traditional sieve, just like our triangle creation procedure, `create_triangle_vertically`. Furthermore, we observe that the Horner-based sieve collects the values of the hypotenuse of the previous block in order to create the next, as made explicit in Formula 11.19, just like the dual sieve, `create_triangles_vertically`. Together, these observations suggest that we can state an equivalence proof between `create_triangle_vertically` and `create_horner_block`.

In order to do so, we take a step back and state an equivalence relation between `create_horner_block` and `create_horner_block_acc`,

```
Theorem create_horner_block_eq_create_horner_block_acc :
  ∀ (n x : nat) (cs : polynomial),
    create_horner_block n x cs =
    create_horner_block_acc n x (cs ++ [0]).
```

since the latter exhibits a structure similar to create_triangle_vertically. We prove the above relation by case analysis on the number of divisions, n, followed by case analysis on the polynomial, cs, and rewriting with the following helper lemma,

```
Lemma removelast_horner_poly_div_acc :
  ∀ (cs : polynomial) (x a : nat),
    removelast (horner_poly_div_acc (cs ++ [0]) x a) =
    horner_poly_div_acc cs x a.
```

which itself is proved by structural induction on the polynomial, cs. Having proved the relation between create_horner_block and create_horner_block_acc, we have taken care of the extra base case for create_horner_block and can now state the following equivalence relation between create_horner_block_acc and create_triangle_vertically,

```
Theorem create_horner_block_acc_eq_create_triangle_vertically :
  ∀ (r : nat) (σ  : Stream nat),
    hypotenuse (create_horner_block_acc
                  r 1 (Str_prefix (S r) σ )) =
    hypotenuse (create_triangle_vertically
                  (tuple_constant (S r) 0) (Str_prefix (S r) σ )).
```

which we prove by induction on the prefix length, r, and rewrite using a set of equivalence relations between list_partial_sums, stream_partial_sums, make_tuple and horner_poly_div. Specifically, we prove the following equivalence relation between list_partial_sums and make_tuple,

```
Corollary equivalence_of_list_partial_sums_and_make_tuple :
  ∀ (xs : tuple),
    make_tuple xs 0 =
    removelast (list_partial_sums xs).
```

that is a corollary of the equivalence relation,

```
Lemma equivalence_of_list_partial_sums_acc_and_make_tuple :
  ∀ (xs : tuple) (a : nat),
    make_tuple xs a =
    removelast (list_partial_sums_acc a xs).
```

which is proved by structural induction on the tuple, xs. Furthermore, we prove the relation,

```
Lemma list_partial_sums_eq_horner_poly_div :
  ∀ (cs : polynomial),
    list_partial_sums cs =
    horner_poly_div cs 1.
```

that is proved by case analysis on the polynomial, cs, and rewriting with the similar relation,

```
Lemma list_partial_sums_acc_eq_horner_poly_div_acc :
  ∀ (cs : polynomial) (a : nat),
    list_partial_sums_acc a cs =
    horner_poly_div_acc cs 1 a.
```

which we prove by structural induction on the polynomial, cs. Finally, we also use the theorem,

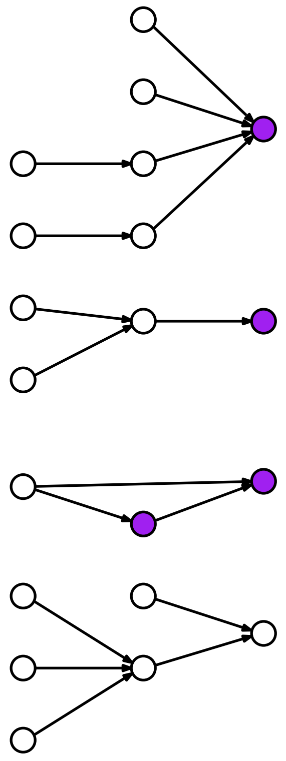
```
Theorem equivalence_of_make_tuple_and_stream_partial_sums_acc :
  ∀ (l' a : nat) (σ  : Stream nat),
    make_tuple (Str_prefix (S l') σ ) a =
    Str_prefix l' (stream_partial_sums_acc a σ ).
```

which we have already proved in Chapter 5. With all these helper lemmas proved we obtain create_horner_block_acc_eq_create_triangle_-vertically, which shows that create_triangle_vertically and create_-horner_block_acc have a simple to state equivalence relation, which captures the fact that the sieve observed by van Yzeren is actually emulating create_-triangles_vertically, which accounts for it being the "rotated mirror image" of Moessner's sieve.

11.4 Summary

In this chapter, we have derived the dual of Moessner's sieve from Horner's method and proved their equivalence.

In order to derive the dual sieve, we first introduced Horner's method for polynomial evaluation and polynomial division, after which we stated the polynomial remainder theorem and Taylor's theorem that we then used to demonstrate how to obtain Taylor polynomials, using Horner's method for polynomial division. As a result, we transformed the successive calculations of Taylor polynomials, called Horner blocks, into the dual sieve we have been working with throughout this dissertation, and proved an equivalence between the two sieve implementations.



Dependency graph of the proofs introduced in Chapter 11. Similarly to the dependency graph of the grid of triangles, we obtain the equivalence between Moessner's sieve and Horner's method using just a small set of new formalizations, reflecting their common foundation.

Chapter 12

Conclusion and perspectives

In the end is my beginning.

T.S. Eliot

One never notices what has been done;
one can only see what remains to be done.

Marie Curie

The goal of this chapter is to conclude our dissertation by summarizing the work we have done while discussing future work.

The chapter is structured as follows. In Section 12.1, we take a retrospective look at this study and we run some statistics on the proof scripts we have written in the process. In Section 12.2, we conclude our findings and we discuss future work in Section 12.3.

12.1 Retrospective

Measuring programming progress by lines of code
is like measuring aircraft building progress by weight.

Bill Gates

In the process of writing this dissertation, we have also written thousands of lines of Coq code ranging from trivial unfolding lemmas to complex theorems. In retrospect, all the statements of our proof scripts exhibit a strikingly regular structure, mainly due to our overall elementary approach. We therefore wrote a Python script that parses the Coq code and performs a range of statistical calculations. Below, we present a breakdown of the keywords and tactics used, followed by some aggregated values showing the size and complexity of our proofs.

Counting words

Table 12.1 presents the number of occurrences of the different keywords and tactics used in our proof scripts. In the first subtable of Table 12.1, there

Keyword	#
Inductive	4
CoInductive	0
Notation+Infix	21
Fixpoint	25
CoFixpoint	17
Definition	34
Theorem	61
Lemma	415
Corollary	73
Instance	18

Tactic	#
rewrite	2819
reflexivity	696
induction	145
bisimilar (coinduction)	61
apply+exact	445
inversion	68
case	108
unfold	300
fold	67
intro(s)	765

Table 12.1: Word count of the keywords and tactics used in our proof scripts.

are exactly 0 `CoInductive` definitions. This is because we import the `Stream` definition from the standard library, and we define `bisimilarity` using an `Inductive` generating function rather than a `CoInductive` proposition, in order to use the paco library for our coinduction proofs. Furthermore, there is a total of 76 `Definitions`, `Fixpoints`, and `CoFixpoints`, reflecting the size of our scaffolding. Likewise, we have proved a total of 567 theorems, lemmas, corollaries, and instances, of which 61 are theorems, suggesting a lower bound on the number of non-trivial properties proved in the context of Moessner's sieve.

The second subtable of Table 12.1, shows the result of sticking to equational reasoning as the core of our reasoning, given the 2819 uses of `rewrite` and 696 uses of `reflexivity`. Furthermore, throughout this dissertation we have performed 145 proofs by induction, 61 proofs by coinduction, and 108 proofs by case analysis. Consequently, the values in Table 12.1 reflect the elementary approach we have taken, focusing on equational reasoning supported by (co)induction proofs and case analysis, where every step of a proof is clearly spelled out.

Aggregated statistics

As mentioned in the previous section, the final proof script of this dissertation consists of 567 proofs. This is also reflected in Table 12.2, which further shows that the 567 proofs consist of 5908 proof steps where the average length of a proof is 10.42 proof steps and the median is 5 proof steps. However, while the average and median values are pretty close to each other there still exists a significant span in the length of the proofs, since the shortest proof is just

number of proofs	567
total number of proof steps	5908
average proof length	10.42 proof steps
median proof length	5 proof steps
min proof size	2 proof steps
max proof size	279 proof steps
compilation time	34.89 seconds
lines of code	14782

Table 12.2: Aggregated statistics over all scripts.

2 steps while the longest is 279 steps. Investigating the reason for the latter value, it turns out that the remarkable length is due to a significant set of nested case analyses. The actual proof behind the value is of the theorem,

```
Theorem equivalence_of_vertical_and_horizontal_triangle_indices :
  ∀ (i j : nat) (xs ys : tuple),
    (length xs) = (length ys) →
    (nth i (nth j (create_triangle_horizontally xs ys) []) 0) =
    (nth j (nth i (create_triangle_vertically xs ys) []) 0).
```

which states the index equivalence of the two triangle creation procedures. The proof was done by nested induction on the entry indices, `i` and `j`, followed by nested case analysis on the two tuples, `xs` and `ys`. This example emphasizes the tendency throughout our proof scripts that long proofs are often a consequence of many cases having to be proved, and not necessarily a sign of complexity.

Lastly, we also investigate the complexity of our script with respect to the compile time of the whole script from scratch. As a result, we find that compiling the 14288 lines of code, divided across 14 different scripts, took half a minute on a MacBook Pro with a 2.3 Ghz i5 processor and 8 GB of memory.

Summary

In this section, we have taken a retrospective look at the actual Coq scripts used in this dissertation and we calculated statistics on them. These calculations were made possible by the regular structure of the proofs, which is the result of our elementary approach to Coq. Consequently, the calculated statistics reflect our large proof base in terms of the number of proofs and proof steps made, but it also reflects the elementary nature of our approach as the majority of proof steps are equational rewrites, (co)induction proofs or case analyses. This approach has also been emphasized throughout the dissertation by the dependency graphs accompanying each chapter, which depicts the interdependence and flow of the lemmas and theorems associated with each chapter.

12.2 Conclusion

In this dissertation, we have characterized Moessner's sieve and proved Moessner's theorem along with some of its generalizations. Specifically, we have formalized the dual of Moessner's sieve that generates a sequence of Moessner triangles, each constructed column by column, for which the bottom-most elements correspond to the traditional result sequence of Moessner's sieve. Furthermore, we have defined a characteristic function of the dual of Moessner's sieve, which we have also proved to be correct, that calculates a specific entry of a Moessner triangle without having to generate the prefix of the sieve. Using these constructs, we have proved Moessner's theorem adapted to the dual sieve, called Moessner's idealized theorem, and generalized it to an initial configuration consisting of a seed tuple with a single seed value of 1, providing a minimal initial configuration for Moessner's theorem.

Going beyond Moessner's theorem, we have introduced a new property of Moessner's sieve that establishes a connection between Moessner triangles of different rank, which suggests the existence of a 2-dimensional grid of Moessner triangles, instead of just a 1-dimensional sequence. Furthermore, we have tested the generality of the dual sieve by stating and proving Long's theorem in terms of it, called Long's idealized theorem, which has led to the conjecture of a new generalization of Long's theorem connecting Moessner's sieve to polynomial evaluation.

Lastly, we have proved an equivalence relation between the repeated application of Horner's method for polynomial division and the dual sieve, which strengthens the relation between Moessner's sieve and polynomial evaluation.

The above findings conclude the work of this dissertation.

12.3 Future work

We plan to prove the conjecture stated in Chapter 10 that generalizes the initial configuration of Long's idealized theorem to an arbitrary seed tuple of constants. We also wish to further explore the possibilities and properties of the grid of Moessner triangles discussed in Chapter 9. Lastly, we wish to explore the possible link between Moessner's sieve and the range of algebraic concepts such as Stirling numbers of the second kind [13], difference sequences, Babbage's Difference engine [36] and polynomial root finding [33], which are all related to Horner's method.

Bibliography

[1] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The Coq proof assistant reference manual–version 7.2. Technical Report 255, INRIA, Paris, 2002. (Cited on pages 11 and 16.)

[2] Yves Bertot. Filters on CoInductive Streams, an Application to Eratosthenes' Sieve. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer, Heidelberg, 2005. (Cited on page 11.)

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*. Springer, Heidelberg, 2004. (Cited on pages 11 and 16.)

[4] Mark Bickford, Dexter Kozen, and Alexandra Silva. Formalizing Moessner's theorem in Nuprl. `http://www.nuprl.org/documents/Moessner/`, August 2011. (Cited on page 10.)

[5] Mark Bickford, Dexter Kozen, and Alexandra Silva. Formalizing Moessner's Theorem and Generalizations in Nuprl. In Ralph Matthes and Aleksy Schubert, editors, *Types 2013 Post-proceedings*, pages 1–7. Leibniz International Proceedings in Informatics, 2013. (Cited on page 10.)

[6] Florian Cajori. Horner's method of approximation anticipated by Ruffini. *Bulletin of the American Mathematical Society*, 17(8):409–414, November 1911. (Cited on pages 9 and 127.)

[7] Christian Clausen, Olivier Danvy, and Moe Masuko. A characterization of Moessner's sieve. *Theoretical Computer Science*, 2014. DOI: 10.1016/j.tcs.2014.03.012. (Cited on pages 11, 13, 45, 69, and 164.)

[8] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and computation*, 76(2):95–120, 1988. (Cited on page 16.)

[9] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972. (Cited on page 16.)

[10] Peter Fox. *Cambridge University Library: the great collections*. Cambridge University Press, 1998. (Cited on page 52.)

[11] George Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008. (Cited on page 16.)

[12] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, 1989. (Cited on page 10.)

[13] Tian-Xiao He and Peter J.-S. Shiue. A Note on Horner's Method. *Journal of Concrete and Applicable Mathematics*, 10(1):53, January 2012. (Cited on pages 127 and 148.)

[14] Ralf Hinze. Concrete Stream Calculus: An extended study. *Journal of Functional Programming*, 20(5-6):463–535, 2010. (Cited on pages 12, 46, and 68.)

[15] Ralf Hinze. Scans and Convolutions - A Calculational Proof of Moessner's Theorem. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages, 20th International Workshop, IFL 2008*, number 5836 in Lecture Notes in Computer Science, pages 1–24. Springer, Hatfield, UK, September 2011. (Cited on pages 12 and 59.)

[16] William G. Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, 1819. (Cited on pages 9, 127, and 129.)

[17] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–206. ACM, 2013. (Cited on page 19.)

[18] Dexter Kozen and Alexandra Silva. On Moessner's Theorem. Technical report, Computing and Information Science, Cornell University, June 2011. (Cited on page 10.)

[19] Robert Krebbers, Louis Parlant, and Alexandra Silva. Moessner's Theorem: an exercise in coinductive reasoning in Coq, 2013. (Cited on pages 11 and 13.)

[20] Christoph Kreitz. The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide. *Department of Computer Science, Cornell University*, 2002. (Cited on page 10.)

[21] Xavier Leroy. The CompCert C Verified Compiler, 2012. (Cited on page 16.)

[22] Calvin T. Long. On the Moessner Theorem on Integral Powers. *The American Mathematical Monthly*, 73(8):846–851, 1966. (Cited on pages 7, 12, 13, 40, 46, 59, 68, and 114.)

[23] Calvin T. Long. Mathematical Excitement — The Most Effective Motivation. *The Mathematics Teacher*, 75(5):413–415, 1982. (Cited on pages 9 and 13.)

[24] Calvin T. Long. Strike it out: Add it up. *The Mathematical Gazette*, 66(438):273–277, 1982. (Cited on page 8.)

[25] Calvin T. Long. A Note on Moessner's Process. *The Fibonacci Quarterly*, 24(4):349–356, November 1986. (Cited on page 8.)

[26] Alexander Markowich Ostrowski. On Two Problems in Abstract Algebra Connected with Horner's Rule. In *Studies in Mathematics and Mechanics presented to Richard von Mises*, pages 40–48, New York, 1954. (Cited on pages 128 and 129.)

[27] Alfred Moessner. Eine Bemerkung über die Potenzen der natürlichen Zahlen. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse*, 29(3):9, March 1951. (Cited on pages 1, 6, and 164.)

[28] Milad Niqui and Jan J.M.M. Rutten. Sampling, Splitting and Merging in Coinductive Stream Calculus. In *Mathematics of Program Construction*, pages 310–330. Springer, 2010. (Cited on page 10.)

[29] Milad Niqui and Jan J.M.M. Rutten. A proof of Moessner's theorem by coinduction. *Higher-Order and Symbolic Computation*, 24(3):191–206, 2011. (Cited on pages 11, 12, and 13.)

[30] Milad Niqui and Jan J.M.M. Rutten. An exercise in coinduction: Moessner's theorem. Technical Report 1103, CWI Amsterdam, 2011. (Cited on pages 11 and 12.)

[31] Milad Niqui and Jan J.M.M. Rutten. Stream processing coalgebraically. *Science of Computer Programming*, 78(11):2192–2215, 2013. (Cited on page 10.)

[32] Ivan Paasche. Ein neuer Beweis der Moessnerschen Satzes. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse*, 30(1):1–5, Feburary 1952. (Cited on page 7.)

[33] Alex Pathan and Tony Collyer. The wonder of Horner's method. *The Mathematical Gazette*, 87(509):230–242, July 2003. (Cited on pages 127, 137, and 148.)

[34] Oskar Perron. Beweis des Moessnerschen Satzes. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse*, 29(4):31–34, May 1951. (Cited on page 7.)

[35] Karel A. Post. Moessnerian theorems. How to prove them by simple graph theoretical inspection. *Elemente der Mathematik*, (2):46–51, 1990. (Cited on page 10.)

[36] Charles L. Rino. A Mathematical Tour of Babbage Difference Engine No. 2. `http://chuckrino.com/wordpress/wp-content/uploads/2011/05/BabbageDE2NotesVER1.pdf`, November 2011. (Cited on page 148.)

[37] Jan J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. (Cited on page 10.)

[38] Jan J.M.M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005. (Cited on pages 10 and 28.)

[39] Jan J.M.M. Rutten. A tutorial on coinductive stream calculus and signal flow graphs. *Theoretical Computer Science*, 343(3):443–481, 2005. (Cited on page 12.)

[40] Hans Salié. Bemerkung zu einem Satz von A. Moessner. *Aus den Sitzungsberichten der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse*, 30(2):7–11, Februray 1952. (Cited on page 7.)

[41] Saed Samadi, M. Omair Ahmad, and M. N. S. Swamy. Multiplier-Free Structures for Exact Generation of Natural Powers of Integers. In *International Symposium on Circuits and Systems (ISCAS 2005)*, pages 1146–1149, Kobe, Japan, May 2005. IEEE. (Cited on page 10.)

[42] John G. Slater. Strike it out - some exercises. *The Mathematical Gazette*, 67(442):288–290, 1983. (Cited on page 8.)

[43] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013. (Cited on page 16.)

[44] Jan van Yzeren. A Note on an Additive Property of Natural Numbers. *The American Mathematical Monthly*, 66(1):53–54, January 1959. (Cited on pages 9, 13, and 140.)

[45] Victor Ya Pan. Methods of computing values of polynomails. *Russian Mathematical Surveys*, 21(1):105, 1966. (Cited on pages 128 and 129.)

Glossary

algebra An algebra is a mathematical structure defined over two sets, R and M, each having two binary operations called addition and multiplication. Furthermore, an operation exists, called scalar multiplication, which is defined over both R and M. The operations of R and M satisfy the following axioms:

- Addition $(+)$ over R satisfies:
  - Associativity: $\forall a, b, c \in R, (a + b) + c = a + (b + c)$.
  - Commutativity: $\forall a, b \in R, a + b = b + a$.
  - Identity: $\exists id \in R, \forall a \in R, a + id = a \wedge id + a = a$.
- Multiplication $(\cdot)$ over R satisfies:
  - Associativity: $\forall a, b, c \in R, (a \cdot b) \cdot c = a \cdot (b \cdot c)$.
  - Commutativity: $\forall a, b \in R, a \cdot b = b \cdot a$.
  - Identity: $\exists id \in R, \forall a \in R, a \cdot id = a \wedge id \cdot a = a$.
  - Zero: $\exists z \in R, \forall a \in R, a \cdot z = z \wedge z \cdot a = z$.
- Multiplication distributes over addition:
  - Left distributivity: $\forall a, b, c \in R, a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.
  - Right distributivity: $\forall a, b, c \in R, (b + c) \cdot a = (b \cdot a) + (c \cdot a)$.
- All the axioms above also holds true for addition and multiplication over M.
- Scalar multiplication $(\cdot)$ over R and M satisfies:
  - Left distributivity: $\forall r \in R, \forall a, b \in M,$
    $r \cdot (a + b) = (r \cdot a) + (r \cdot b)$.
  - Right distributivity: $\forall r, s \in R, \forall a \in M,$
    $(r + s) \cdot a = (r \cdot a) + (s \cdot a)$.
  - Associativity: $\forall r, s \in R, \forall a \in M, (r \cdot s) \cdot a = r \cdot (s \cdot a)$.
  - Identity: $\forall a \in M, id_R \cdot a = a$.
  - Bilinear mapping: $\forall r \in R, \forall a, b \in M,$
    $r \cdot (ab) = (r \cdot a)b = a(r \cdot b)$.

(Not used)

algebraic See algebra. (Pages 5, 6, 13)

binomial A binomial is a polynomial with two non-zero terms,

$$p(x) = a_i x^i + a_j x^j,$$

where x is a variable, a_i and a_j are coefficients, and $i \neq j$. In this dissertation, we mainly focus on the set of binomials that are also polynomials of degree 1, i.e., $p(x) = a_1 x + a_0$. (Pages 9, 54, 122, 123, 130, 132, 133, 135–137)

binomial coefficient Given two natural numbers, n and k, the binomial coefficient, $\binom{n}{k}$, can be read as the coefficient of the kth monomial in the binomial expansion of $(x+y)^n$. The binomial coefficient can be calculated in several ways, one of which is the following recursive formula,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \text{ for } 1 \leq k \leq n,$$

with base cases,

$$\binom{n}{0} = 1, \text{ for all } n \geq 0, \text{ and}$$
$$\binom{0}{k} = 0, \text{ for all } k \geq 1.$$

Lastly, the binomial coefficient can also be read from Pascal's triangle as the kth entry of the nth row. Conversely, the entries of Pascal's triangle can be calculated with the above formula, which is also known as Pascal's rule. (Pages 3, 17, 51, 52, 54–58, 60, 62–64, 68, 70, 72)

binomial expansion The binomial theorem states that any exponentiation of the form $(x+y)^n$, where x and y are variables and n is a natural number, can be expanded into the sum,

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k,$$

where $\binom{n}{k}$ is a binomial coefficient. In this dissertation, we focus on a simpler version where y has been substituted with the constant 1,

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k,$$

yielding a polynomial of degree n, where x is a variable and $\binom{n}{k}$ enumerates its coefficients. (Pages 3, 11, 12, 55, 56, 67–69, 71, 74, 75, 84, 88–90, 92, 100, 101, 117)

binomial theorem Given two natural numbers, x and y, and an exponent, n, the exponentiation $(x+y)^n$ can be expanded into the sum,

$$(x+y)^n = \binom{n}{0} x^n y^0 + \binom{n}{1} x^{n-1} y^1 + \cdots + \binom{n}{n-1} x^1 y^{n-1} + \binom{n}{n} x^0 y^n,$$

where $\binom{n}{k}$ is the binomial coefficient. This expansion can also be written in summation notation as,

$$\begin{aligned}(x+y)^n &= \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k \\ &= \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k},\end{aligned}$$

where the last equivalence follows from the symmetry of the sequence of binomial coefficients and of x and y. (Pages 51, 52, 54, 56, 64, 89)

bisimilar See bisimulation. (Pages 30–33, 75, 95)

bisimilarity See bisimulation. (Pages 18–20, 28, 30–33, 36, 76, 77)

bisimulation A bisimulation is a binary relation, $R \subseteq S \times T$, which relates two coinductive types, S and T, that are observably similar. For example, given a bisimulation over the streams of natural numbers, $R \subseteq \mathbb{N}^\omega \times \mathbb{N}^\omega$, and two streams, σ and τ, then the two streams are said to be bisimilar if their initial values and stream derivatives are indistinguishable. This can be stated as the proposition: $\forall \sigma, \tau \in \mathbb{N}^\omega$,

$$(\sigma, \tau) \in R \Rightarrow \begin{cases} (1) & \sigma(0) = \tau(0) \text{ and} \\ (2) & (\sigma', \tau') \in R. \end{cases}$$

Lastly, since the bisimilarity relation, R, is an equivalence relation, we want to reference it with an infix symbol. Hence, we use the symbol '$\sim$' that gives us the notation $\sigma \sim \tau$, instead of $(\sigma, \tau) \in R$. (Pages 30, 31)

case analysis Case analysis, or proof by exhaustion, is a proof technique in which the proof of a statement is split into a finite number of cases, each of which is proved separately. For example, we may prove a statement over natural numbers by splitting it into a proof where $n = 0$ and one where $n = S\ n'$. (Pages 44, 57, 58, 61, 62, 72, 80, 83, 104, 119, 133, 141, 146, 147)

characteristic function A characteristic function describes every result of a procedure, without emulating the procedure itself. The running example of this dissertation is a characteristic function of Moessner's sieve, which can calculate any entry for a given Moessner triangle, without needing to compute the prefix of the sieve first. (Pages 2, 3, 67–70, 72–76, 78, 82, 84, 87, 88, 99, 105, 107, 109, 148)

coalgebra A coalgebra is a mathematical structure defined over an inductive set R and a coinductive set M, each having two binary operations called addition $(+)$ and multiplication $(\cdot)$. Furthermore, an operation exists, called scalar multiplication $(\cdot)$, which is defined over both R and M. The sets R and M, their operations and scalar multiplication satisfies the same axioms as in the case of algebras. (Page 10)

coalgebraic See coalgebra. (Pages 5, 10, 13)

coinduction Coinduction is the mathematical dual of structural induction used for proving properties of coinductive types, i.e., streams, in contrast to inductive types, i.e., lists. One of the techniques used for proving properties of coinductive types is the construction of a binary relation know as a bisimulation, which relates two coinductive types that are behaviorally equivalent. (Pages 12, 17, 19, 20, 30–33, 73, 75, 95, 121, 146)

coinduction principle The coinduction principle states that if two streams are bisimilar then they are also element-wise equal: $\forall \sigma, \tau \in \mathbb{N}^{\omega}$,

$$\sigma \sim \tau \Rightarrow \forall i, \sigma(i) = \tau(i).$$

(Pages 28, 29, 32, 33)

coinductive See coinduction. (Pages 5, 10–13, 18, 19, 21, 28, 33, 36)

coinductive type Coinductive types are the dual of inductive types, which are defined using coinduction instead of induction. Coinductive types represent infinite data structures. (Pages 17, 18, 28)

Coq proof assistant The Coq proof assistant is an interactive theorem prover that supports the development of mathematical proofs and formal specifications in a computational setting. All programs, properties, and proofs are formalized in the Calculus of Inductive Constructions (CIC) and verified by a type checking algorithm. (Pages 3, 11, 15, 16, 20, 21, 56, 58, 60, 107)

corecursion Corecursion is the dual of recursion. While recursion consumes data by breaking down elements of an inductive type into its subparts until reaching a base case, corecursion produces data of a coinductive type. For example, a corecursive procedure may lazily construct a stream given a seed value and a progress function for generating the tail of the stream. (Not used)

corecursive See corecursion. (Pages 11, 18, 29)

dropped sequence The dropped sequence is the sequence consisting of the elements dropped in one iteration of Moessner's sieve. For example, when applying Moessner's sieve on the sequence of natural numbers,

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \dots,$$

with rank $k = 2$, the dropped sequence becomes

$$2, 4, 6, 8, 10, \ldots,$$

while the result sequence becomes

$$1, 4, 9, 16, 25, \ldots.$$

(Page 9)

dual sieve The dual sieve is an abbreviation for the "dual of Moessner's sieve", which constructs a sequence of Moessner triangles generated column by column, as opposed to a result sequence generated row by row. (Pages 2, 3, 48, 67, 79, 87, 95, 113, 114, 116, 121, 123, 124, 140, 142, 148)

element An element is a specific member of a collection, e.g., a sequence or list. For example, the value 3 is the third element of the list $[1, 2, 3, 4]$. (Pages 23–26, 29, 30, 40)

element-wise equal See element-wise equality. (Pages 32, 77)

element-wise equality Element-wise equality is the property of two ordered collections having the exact same members. As such, we say that two streams are element-wise equal if, $\forall \sigma, \tau \in \mathbb{N}^\omega$,

$$\forall i \in \mathbb{N}, \sigma(i) = \tau(i).$$

(Pages 32, 33, 76)

equational reasoning Equational reasoning is the act of reasoning about a proof or program by substituting expressions using existing equivalences. For example, if we know that $y = 3$ and $x = 5$, we can rewrite the expression $y^2 = 2x - 1$ like so,

$$\begin{aligned} y^2 &= 2 \cdot x - 1 \\ 3^2 &= 2 \cdot x - 1 \\ 3^2 &= 2 \cdot 5 - 1 \\ 9 &= 2 \cdot 5 - 1 \\ 9 &= 10 - 1 \\ 9 &= 9, \end{aligned}$$

thus obtaining a proposition we know to be true by reflexivity of Leibniz equality. (Pages 16, 20, 146)

Eratosthenes' sieve Eratosthenes' sieve is a procedure for finding all prime numbers up to a given limit. The essence of the procedure is to iteratively mark numbers which are multiples of already seen values, starting with multiples of 2, leaving the primes unmarked. (Page 11)

finite sequence A finite sequence is a sequence with a length, $n \in \mathbb{N}$,

$$a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}, a_n,$$

in contrast to an infinite sequence which continues indefinitely. (Not used)

finite series A finite series is a series with a defined first and last term,

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + a_2 + \cdots + a_{n-1} + a_n,$$

where $n \in \mathbb{N}$, in contrast to an infinite series which continues indefinitely. (Not used)

generating function Generating functions are synonymous with power series. (Page 12)

guardedness condition The guardedness condition captures the two restrictions made on recursive and corecursive calls in the Coq proof assistant. For recursive calls, the argument being passed must be a subpart of the initial argument. Conversely, every corecursive call must be an argument of a coinductive type constructor. (Page 11)

Horner block A Horner block is the resulting triangular array of values constructed by repeatedly dividing a polynomial, and its quotients, with a binomial, $x - k$, using Horner's method for polynomial division. For example, performing the procedure on the polynomial,

$$f(x) = 1x^4 + 0x^3 + 0x^2 + 0x^1 + 0x^0,$$

where $k = 1$, yields the following Horner block,

$$\begin{array}{ccccc}
1 & 0 & 0 & 0 & 0 \\
 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & \mathbf{1} \\
 & 1 & 2 & 3 & \\
1 & 2 & 3 & \mathbf{4} & \\
 & 1 & 3 & & \\
1 & 3 & \mathbf{6} & & \\
 & 1 & & & \\
\mathbf{1} & \mathbf{4} & & &
\end{array}$$

where the hypotenuse, $(1, 4, 6, 4, 1)$, describes the coefficients of the Taylor polynomial $F_{4,1}$, where 4 is the degree of the polynomial and 1 is the value of k. (Pages 9, 10, 137, 139, 140, 142)

Horner's method Horner's method is a pair of efficient procedures for evaluating or dividing a polynomial using the same recursive substitution scheme. (Pages 3, 9, 13, 123, 127–130, 132–135, 137–139, 142, 148)

hypotenuse A hypotenuse is the longest side in a right-angled triangle. In this dissertation, we use 'hypotenuse' to describe the last diagonal of a Moessner triangle or Horner block. (Pages 9, 11, 12, 47, 49, 68, 69, 74, 76, 77, 87–93, 95, 96, 101, 104, 106, 118, 122, 137–140)

induction Induction, specifically structural induction, is a proof technique for proving that some proposition $P(x)$ holds for all x of some inductive type A. For every inductive type A there exists a partial order, $<$, such that proofs by structural induction work by first proving that the proposition P holds for all ground structures of A, called the base cases, followed by proving that it also holds for all composite structures, called the inductive cases. If P holds for all base cases and inductive cases, then it also holds for all x. Note that we can use this approach to perform mathematical induction over natural numbers, by doing structural induction over an inductive `nat` type. (Pages 16, 17, 20, 25, 28, 30, 32, 33, 42, 44, 57, 58, 61, 62, 72, 76, 77, 79–81, 83, 88, 89, 91–94, 104, 108, 118–120, 133, 141, 142, 146, 147)

inductive See induction. (Pages 5, 6, 13, 16, 17, 21, 23–25, 30, 33, 52–54, 56, 59, 60, 62, 63, 71, 74, 77–79, 102, 118, 128, 129)

inductive type Inductive types are the dual of coinductive types, which are defined using induction instead of coinduction. Inductive types represent finite data structures, e.g., lists. (Pages 17, 18, 23, 63)

infinite sequence An infinite sequence is a sequence which continues indefinitely,

$$a_0, a_1, a_2, \ldots,$$

in contrast to a finite sequence with a specific length. (Not used)

infinite series An infinite series is a series which continues indefinitely,

$$\sum_{i=0}^{\infty} a_i = a_0 + a_1 + a_2 + \cdots,$$

in contrast to a finite series with a defined first and last term. (Not used)

initial configuration An initial configuration is the set of objects from which an instance of Moessner's sieve is generated. For example, for an instance of the dual sieve, the initial configuration is a horizontal and vertical seed tuple, while it is an initial sequence for the traditional version of Moessner's sieve. (Pages 2, 3, 39, 45, 46, 48, 49, 114, 115, 119, 122, 123, 148)

initial sequence The initial sequence is the sequence on which the first iteration of Moessner's sieve is applied. (Pages 1, 2, 6–8, 12, 34, 35, 114)

initial value See stream. (Pages 10, 18, 28–31, 33, 35, 72, 75, 95, 121)

Leibniz equality Leibniz equality is the smallest equivalence relation between two objects, x and y, written $x = y$. Thus, if we can reduce y to x, or conversely, then x and y are equivalent with respect to Leibniz equality. (Pages 18, 30, 31, 33, 36)

list A list is an inductive type representing a finite sequence, defined like so:

- The empty list, [], is a list.
- Given a list, l', we can construct a new list by adding an element, e, onto it, $e :: l'$.

(Pages 6, 17, 23–28, 30, 33, 34, 36, 42, 44, 45, 47–49, 92, 93, 102, 118, 119, 129, 130, 132, 138)

list calculus We define a list calculus to be an inductive list type together with a set of selectors, constructors and operators which manipulate lists. (Pages 3, 23, 27–29, 34, 36)

Long's idealized theorem Long's idealized theorem states that applying the dual sieve on a vertical seed tuple containing a pair of constants, c and d, followed by 0s, yields the result sequence,

$$c \cdot 1^{k-2} + d \cdot 1^{k-3}, c \cdot 2^{k-2} + d \cdot 2^{k-3}, c \cdot 3^{k-2} + d \cdot 3^{k-3}, \dots,$$

where $k > 2$ is the length of the vertical seed tuple. (Pages 2, 3, 113, 115–118, 121–124, 148)

Long's theorem Long's (original) theorem states that applying Moessner's sieve on the initial sequence,

$$a, a + d, a + 2d, a + 3d, \dots,$$

yields the result sequence,

$$a \cdot 1^{k-1}, (a + d) \cdot 2^{k-1}, (a + 2d) \cdot 3^{k-1}, \dots,$$

where k is the rank of the sieve. (Pages 2, 3, 113, 114, 116, 124, 148)

Long's weak theorem Long's weak theorem states that applying the dual sieve on a vertical seed tuple consisting of a constant, c, followed by 0s, yields the result sequence,

$$c \cdot 1^{k-2}, c \cdot 2^{k-2}, c \cdot 3^{k-2}, \dots,$$

where $k > 2$ is the length of the vertical seed tuple. (Pages 113, 116–118, 120, 122, 124)

Moessner triangle A Moessner triangle is one of the resulting triangular arrays of values generated when applying Moessner's sieve on a given sequence. For example, when applying Moessner's sieve of rank 5 on the sequence of 1s, the first Moessner triangle becomes,

$$\begin{array}{ccccc}
1 & 1 & 1 & 1 & \mathbf{1} \\
1 & 2 & 3 & \mathbf{4} & \\
1 & 3 & \mathbf{6} & & \\
1 & \mathbf{4} & & & \\
\mathbf{1} & & & &
\end{array}$$

where the rank of the Moessner triangle is 4, i.e., its depth minus 1. (Pages 2, 3, 7, 8, 10, 39, 40, 42–49, 59, 67–77, 79, 82, 84, 87–90, 92–96, 99, 100, 104–108, 110, 115, 118, 119, 121–123, 148)

Moessner's idealized theorem Moessner's idealized theorem states that applying the dual sieve on an initial configuration consisting of a horizontal seed tuple of rank k (the tuple's length minus 2), filled with 0s, and a vertical seed tuple of rank k, filled with a 1 followed by 0s, yields a sequence of Moessner triangles, where the bottom-most element of each triangle enumerate the sequence of successive powers of rank k. (Pages 2, 3, 87, 93–96, 113, 116–118, 121, 122, 124, 148)

Moessner's sieve Moessner's sieve is a procedure for iteratively dropping elements from a sequence and partially summing the remaining elements. Specifically, given a sequence and a natural number k, called the rank, the procedure first drops every kth element of the initial sequence and partially sums the remaining elements to create a new sequence. From the newly created sequence every $(k-1)$th element is dropped and a new sequence is created through partial summation. The procedure stops when k reaches 1. For example, given the sequence of natural numbers,

$$1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots,$$

and letting $k = 3$. Moessner's sieve generates the following triangular arrays,

$$\begin{array}{cccccccccc}
1 & 2 & \mathbf{3} & 4 & 5 & \mathbf{6} & 7 & 8 & \mathbf{9} & \ldots \\
1 & \mathbf{3} & & 7 & \mathbf{12} & & 19 & \mathbf{27} & & \ldots \\
\mathbf{1} & & & \mathbf{8} & & & \mathbf{27} & & & \ldots
\end{array}$$

where the result sequence located at the last row is the sequence of successive powers,

$$1^k, 2^k, 3^k, \ldots$$

with $k = 3$. For the example above, the result sequence is,

$$1^3, 2^3, 3^3, \ldots,$$

which corresponds to the values $(1, 8, 27, \dots)$. The procedure was originally described by Alfred Moessner [27], while the term 'Moessner's Sieve' was coined by Danvy et al. [7], in reference to Erastosthenes' Sieve. (Pages 1–3, 5–13, 16, 23, 27, 34–36, 39–41, 43, 45–49, 51, 58, 59, 63, 64, 67–69, 72, 74, 78, 82, 84, 87–90, 95, 96, 99–101, 110, 113–115, 122–124, 127, 138, 140, 142, 146, 148)

Moessner's theorem Moessner's (original) theorem states that applying Moessner's sieve on the initial sequence of natural numbers,

$$1, 2, 3, \dots,$$

with rank k yields the result sequence of successive powers

$$1^k, 2^k, 3^k, \dots.$$

(Pages 1–3, 5, 7–13, 72, 87, 95, 148)

monomial A monomial is a polynomial with one non-zero term $a_i x^i$,

$$p(x) = a_i x^i,$$

where x is a variable and a_i is a coefficient. (Pages 11, 12, 54–56, 68–71, 74, 75, 77, 84, 88, 90, 92, 104, 117, 119, 121)

Pascal's rule Pascal's rule is a combinatorial identity for binomial coefficients. Given two natural numbers, n and k, the following relation holds,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \text{ for } 1 \leq k \leq n,$$

where $\binom{n}{k}$ is the binomial coefficient. Alternatively, this relation can be written as,

$$\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}, \text{ for } 1 \leq k \leq n+1.$$

(Pages 53, 57, 61, 68, 69, 71, 73, 89, 101, 104, 106)

Pascal's triangle Pascal's triangle is a triangular array of binomial coefficients,

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
\mathbf{1} & & \mathbf{4} & & \mathbf{6} & & \mathbf{4} & & \mathbf{1}
\end{array}$$

indexed in terms of its rows and their entries, both index from 0. For example, the entry having value 6 in the triangle above is the third entry of the fifth row, thus the binomial coefficient $\binom{4}{2}$ is equal to 6. Pascal's triangle is inductively constructed, starting from the base case of a single entry 1 in the first row, we can construct the next row by adding the two values immediately above the desired entry. For example, entry $(3, 1)$ is created by adding the values of $(2, 1)$ and $(2, 2)$,

$$\begin{tikzcd}
1 \arrow[dr] & & 2 \arrow[dl] \\
& 3 &
\end{tikzcd}$$

(Pages 3, 7, 8, 46, 49, 51–60, 62–64, 68, 71, 72, 101)

polynomial A polynomial, p, is a sum of terms, $a_i x^i$, each consisting of a coefficient, $a_i \in \mathbb{N}$, and a variable, x,

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0.$$

which can be written in summation notation as,

$$p(x) = \sum_{i=0}^{n} a_i x^i.$$

(Pages 9, 123, 127–130, 132–135, 137–139, 141, 142, 148)

polynomial division Polynomial division is the act of dividing a polynomial, p, with another polynomial, d, of the same or lower degree. When dividing two polynomials, p and d, $\frac{p(x)}{d(x)}$, where $d \neq 0$, there exist two polynomials, q and r, such that the result satisfies the relation $p(x) = d(x) \cdot q(x) + r(x)$, where r is either 0 or has a degree less than d. We call p the numerator, d the denominator, q the quotient, and lastly r the remainder. (Pages 3, 127, 128, 130, 132, 133, 137, 138, 142, 148)

polynomial evaluation Given a polynomial,

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

and a natural number k, we evaluate $p(k)$ by replacing all occurrences of x with k, followed by evaluating each of the terms and summing the result. (Pages 3, 123, 124, 127–133, 142, 148)

polynomial long division Polynomial long division is a generalization of traditional long division that makes it possible to divide a polynomial with another polynomial of the same degree or lower. As an example, we let $p(x) = 2x^3 + 4x^2 + 11x + 3$ be the numerator, $d(x) = x - 2$ the denominator, such that we divide p by d using the following procedure:

1. Reorder the numerator, $2x^3 + 4x^2 + 11x + 3$, and denominator, $x - 2$, into the following format,

$$\begin{array}{r|rrrr}
x-2) & 2x^3 & +4x^2 & +11x & +3
\end{array}$$

2. Divide the first term of the numerator, $2x^3$, with the first term of the denominator, x, and write the result, $2x^2$, on the line above the numerator,

$$\begin{array}{r|rrrr}
 & & 2x^2 & & \\ \hline
x-2) & 2x^3 & +4x^2 & +11x & +3
\end{array}$$

3. Multiply the result of the previous step, $2x^2$, with the denominator, $x - 2$, and put the result, $2x^3 - 4x^2$, below the numerator, flipping the sign of each term,

$$\begin{array}{r|rrrr}
 & & 2x^2 & & \\ \hline
x-2) & 2x^3 & +4x^2 & +11x & +3 \\
 & -2x^3 & +4x^2 & & \\ \cline{2-3}
\end{array}$$

4. Subtract the intermediate result of the previous step, $2x^3 - 4x^2$, from the numerator, $2x^3 + 4x^2 + 11x + 3$, and write the result, $8x^2 + 11x$, below,

$$\begin{array}{r|rrrr}
 & & 2x^2 & & \\ \hline
x-2) & 2x^3 & +4x^2 & +11x & +3 \\
 & -2x^3 & +4x^2 & & \\ \cline{2-3}
 & & 8x^2 & +11x &
\end{array}$$

5. Repeat Steps $2 - 4$, now using the intermediate result from Step 4, $8x^2 + 11x$, as the new numerator, until reaching the last term,

$$\begin{array}{r|rrrr}
 & & 2x^2 & +8x & +27 \\ \hline
x-2) & 2x^3 & +4x^2 & +11x & +3 \\
 & -2x^3 & +4x^2 & & \\ \cline{2-3}
 & & 8x^2 & +11x & \\
 & & -8x^2 & +16x & \\ \cline{3-4}
 & & & 27x & +3 \\
 & & & -27x & +54 \\ \cline{4-5}
 & & & & 57
\end{array}$$

6. The resulting quotient, $2x^2 + 8x + 27$, can now be read from the line above the numerator, $2x^3 + 4x^2 + 11x + 3$, and the remainder, 57, can be read from the value below the last horizontal line of the calculation.

In order to verify the calculations above, we simply check that,

$$2x^3 + 4x^2 + 11x + 3 = (x-2)(2x^2 + 8x + 27) + 57,$$

is true. (Pages 130, 132)

polynomial remainder theorem Given a polynomial,

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

where $a_0, \ldots, a_n \in \mathbb{N}$, and a binomial,

$$d(x) = x - k,$$

where $k \in \mathbb{N}$, the remainder of dividing p with d, denoted r, is equal to $p(k)$. Furthermore, d divides p if and only if $p(k) = 0$. (Pages 133, 142)

positive natural numbers The set of positive natural numbers is the ordered set of natural numbers having 1 as the smallest value: $\{1, 2, 3, 4, 5, \ldots\}$. (Pages 1, 2, 6–8, 11, 12)

power series A power series is an infinite series defined over a variable x, having the form,

$$\sum_{i=0}^{\infty} a_i (x-k)^i = a_0 (x-k)^0 + a_1 (x-k)^1 + a_2 (x-k)^2 + \cdots ,$$

where $a_i \in \mathbb{N}$ are the coefficients and $k \in \mathbb{N}$ is a constant. The power series is said to be centered at k. A common power series is the Taylor series,

$$\sum_{i=0}^{\infty} \frac{p^{(i)}(k)}{i!}(x-k)^i = \frac{p(k)}{0!}(x-k)^0 + \frac{p'(k)}{1!}(x-k)^1 + \frac{p''(k)}{2!}(x-k)^2 + \cdots ,$$

where p is a polynomial and the coefficients of the power series are defined in terms of p's derivatives at the point k. When k is equal to zero, the power series has the form,

$$\sum_{i=0}^{\infty} a_i x^i = a_0 x^0 + a_1 x^1 + a_2 x^2 + \cdots ,$$

an example of which is the Maclaurin series, a special case of the Taylor series,

$$\sum_{i=0}^{\infty} \frac{p^{(i)}(0)}{i!} x^i = \frac{p(0)}{0!} x^0 + \frac{p'(0)}{1!} x^1 + \frac{p''(0)}{2!} x^2 + \cdots ,$$

where p again is a polynomial. (Page 10)

procedure A procedure is a sequence of instructions to perform a specific task. A procedure can take a given number of arguments which determine the execution of the procedure. A common procedure in this dissertation is Moessner's sieve, since it drops and partially sums sequences of values in a manner determined by its arguments: its rank and initial sequence. (Pages 1, 2, 6, 11, 18, 19, 23, 26, 27, 29–31, 34–36, 39–47, 49, 67, 74, 75, 78, 79, 84, 90, 92, 93, 95, 96, 100, 102, 103, 105, 107, 110, 127, 129–133, 136, 137, 139)

quotient See polynomial division. (Pages 9, 130–132, 135, 137)

rank The rank of an object describes the size of the resulting Moessner triangles generated from the associated sieve. As such, we define the rank of the following types of objects:

1. The rank of an application of Moessner's sieve is the drop index, often denoted k.
2. The rank of a Moessner triangle is the depth of the triangle minus 1, which is also equal to the drop index minus 1.
3. The rank of a seed tuple is the length of the seed tuple minus 2, which is also equal to the rank of the resulting Moessner triangle.

(Pages 2, 3, 6, 11, 12, 35, 36, 40, 46, 58, 59, 68, 71, 72, 74–76, 88–91, 95, 99–108, 110, 114, 115, 118, 119)

rank decomposition Rank decomposition is the act of describing an entry of a Moessner triangle with triangle index t and rank k, in terms of nearby entries in the same Moessner triangle with triangle index t and rank $k-1$. (Pages 99, 105–108, 110)

rank-upgrading procedure Given a seed tuple of rank k, a rank upgrading procedure returns the corresponding seed tuple of rank $k+1$. (Pages 99, 100, 102, 103, 110)

recursion Recursion is the process of consuming inductively defined data by decomposing it into its subparts, the inductive cases, until reaching an atom, base case. For example, traversing an inductively defined list, l, is done by decomposing the list into a head element, e, and a tail, l', followed by traversing the tail, l', until reaching the empty list, $[]$. In this traversal, the empty list is the base case while the decomposition of the list, l, into a head, e, and a tail, l', is the inductive case. (Not used)

recursive See recursion. (Pages 17, 26, 35, 36, 43, 102, 127, 129–131)

remainder See polynomial division. (Pages 130, 131, 134, 135)

result sequence The result sequence is the sequence resulting from applying Moessner's sieve on an initial sequence. For example, the result sequence of applying Moessner's sieve of rank 2 on the initial sequence of natural numbers,

$$1, 2, 3, \ldots,$$

yields the result sequence of squares,

$$1^2, 2^2, 3^2, \ldots.$$

(Pages 1, 2, 6, 8, 9, 11, 12, 34, 35, 95, 114, 115, 121–123, 148)

result stream A result stream is the same as a result sequence, but within the context of streams. (Pages 36, 40, 48)

rotated binomial coefficient The rotated binomial coefficient describes the entries of the rotated Pascal's triangle, indexed by its rows and columns and containing the same elements as Pascal's triangle. (Pages 3, 51, 58, 60–62, 64)

rotated Pascal's triangle The rotated Pascal's triangle is a left-aligned version of Pascal's triangle,

$$\begin{array}{ccccc}
1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & \\
1 & 3 & 6 & & \\
1 & 4 & & & \\
1 & & & &
\end{array}$$

being indexed by its rows and columns, and having the same entries as Pascal's triangle. (Pages 3, 51, 58–60, 62, 64, 68, 73)

seed stream A seed stream is the same as an initial sequence, but within the context of streams. (Pages 35, 40, 49)

seed tuple A seed tuple is one of the tuples used to create a Moessner triangle with the dual of Moessner's sieve. For example, in the following case,

$$\begin{array}{c|cccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & \\
1 & 1 & 1 & 1 & 1 & 1 & \\
0 & 1 & 2 & 3 & 4 & & \\
0 & 1 & 3 & 6 & & & \\
0 & 1 & 4 & & & & \\
0 & 1 & & & & & \\
0 & & & & & &
\end{array}$$

the vertical seed tuple is $(1, 0, 0, 0, 0, 0)$ while the horizontal seed tuple is $(0, 0, 0, 0, 0, 0)$. Since the horizontal seed tuple is mostly 0s throughout the examples of this dissertation, we often use seed tuple as short for the vertical seed tuple. The rank of a seed tuple is the length of the seed tuple minus 2, which is also equal to the rank of the resulting Moessner triangle. (Pages 2, 3, 39–43, 45–49, 67, 74–76, 79–81, 87, 90, 91, 93, 94, 99–104, 106, 107, 110, 113–115, 117–119, 122–124, 148)

sequence A sequence is a possibly infinite linearly ordered collection whose members are called elements. A sequence is ordered on the index values of its members. The number of elements in a sequence is called the length of the sequence. Two common sequences in this dissertation are the sequence of positive natural numbers,

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \dots,$$

and the sequence consisting of a 1 followed by 0s,

$$1, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots.$$

(Pages 2, 3, 6, 8–12, 34–36, 39, 48, 69, 91, 115, 123, 148)

sequence of partial sums A sequence of partial sums, $\{S_k\}$, defined over a series and a natural number k, is the sum of the sequence $\{a_i\}$ from a_0 to a_k,

$$S_k = \sum_{i=0}^{k} a_i = a_0 + a_1 + a_2 + \cdots + a_{k-1} + a_k.$$

(Not used)

sequence of successive powers The sequence of successive powers is a sequence of positive natural numbers raised to a fixed positive natural number. For example, the sequence of successive powers for the value 3 is the sequence of cubes,

$$1^3, 2^3, 3^3, 4^3, \dots.$$

(Pages 1, 6, 8, 10)

series A series is the sum of the terms of a sequence. A series is finite if it has a defined first and last term, otherwise it is infinite. Given a finite sequence, $\{a_i\}$, the series of the sequence is the sum,

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + a_2 + \cdots + a_{n-2} + a_{n-1} + a_n,$$

where $a_i, n \in \mathbb{N}$, and conversely, given an infinite sequence $\{a_i\}$, the series of the sequence is the sum,

$$\sum_{i=0}^{\infty} a_i = a_0 + a_1 + a_2 + \cdots,$$

where $a_i \in \mathbb{N}$. (Not used)

stream A stream is the computational representation of an infinite sequence. As such, the set of all streams of natural numbers is defined as,

$$\mathbb{N}^\omega = \{\sigma \mid \sigma : \mathbb{N} \to \mathbb{N}\},$$

where,

$$\sigma = (\sigma(0), \sigma(1), \sigma(2), \dots).$$

The entry $\sigma(0)$ is called the initial value of the stream, while the remainder of the stream is called the stream derivative, written,

$$\sigma' = (\sigma(1), \sigma(2), \sigma(3), \dots).$$

(Pages 3, 10, 12, 13, 17–19, 23, 27–36, 39, 40, 42, 43, 45, 46, 48, 49, 67, 68, 73, 75, 77, 78, 82, 83, 88, 91, 95, 118, 120)

stream calculus A stream calculus is a coinductive stream type together with a set of selectors, constructors and operators which manipulate streams. (Pages 3, 10, 12, 13, 19, 23, 27–29, 34, 36)

stream derivative See stream. (Pages 10, 18, 28–33, 72, 75, 76, 95)

stream of successive powers See sequence of successive powers. (Pages 3, 39, 40, 48, 49, 95)

Taylor polynomial A Taylor polynomial is a Taylor series with a finite number of terms,

$$P_{n,k}(x) = \sum_{i=0}^{n} \frac{p^{(i)}(k)}{i!}(x-k)^i,$$

where $n, k \in \mathbb{N}$. If p is a polynomial, then $P_{n,k}$ calculates the same function as p for all values of k, and $P_{n,k}$ is identical to p in the case where $k = 0$ and n is equal to the degree of the polynomial p. (Pages 127, 133–138, 142)

Taylor series Given a polynomial, p, and a natural number, k, then p can be represented as the power series centered at k,

$$\sum_{i=0}^{\infty} \frac{p^{(i)}(k)}{i!}(x-k)^i = \frac{p(k)}{0!}(x-k)^0 + \frac{p'(k)}{1!}(x-k)^1 + \frac{p''(k)}{2!}(x-k)^2 + \cdots,$$

whose coefficients are defined in terms of p's derivatives at the point k. Such a power series is called the Taylor series of p. (Pages 133, 134)

Taylor's theorem Given a polynomial, p, and two natural numbers, n and k, the k-th order Taylor polynomial, $P_{n,k}$, of p centered at k is,

$$P_{n,k}(x) = \sum_{i=0}^{n} \frac{p^{(i)}(k)}{i!}(x-k)^i.$$

Since p is a polynomial, $P_{n,k}$ calculates the same values as p for all values of k, and in the case where $k = 0$, $P_{n,k}$ is the Maclaurin polynomial identical to p. (Pages 133, 134, 142)

triangle creation procedure Given two seed tuples, a triangle creation procedure creates the corresponding Moessner triangle. In this dissertation, we work with two triangle creation procedures, `create_triangle_horizontally` and `create_triangle_vertically`, which create a Moessner triangle either row by row or column by column. (Pages 3, 39, 42–45, 49, 67, 74, 76, 78, 82, 84, 87, 93, 105, 107, 109, 140, 147)

tuple A tuple is an ordered set of elements with a fixed length n, and usually written within parentheses, $(a_0, a_1 \ldots, a_{n-2}, a_{n-1})$. (Pages 3, 40–42, 44, 47, 49, 78–80, 101, 102, 118–120, 123, 124, 141, 147)

Appendices

Appendix A

Coq Definitions

```
Fixpoint app {A : Type} (xs ys : list A) : list A :=
  match xs with
    | [] ⇒ ys
    | x :: xs' ⇒ x :: app xs' ys
  end.
Infix "++ " := app (right associativity, at level 60) : list_scope.

Fixpoint binomial_coefficient (n k : nat) : nat :=
  match n, k with
    | n, 0 ⇒ 1
    | 0, S k' ⇒ 0
    | S n', S k' ⇒ binomial_coefficient n' (S k') +
                  binomial_coefficient n' k'
  end.
Notation "C( n , k )" := (binomial_coefficient n k).

CoInductive bisimilarity (σ  τ : Stream nat) : Prop :=
  bisimilar : σ (0) = τ (0) →
              σ ' ∼ τ '→
              σ ∼ τ
where "σ  ∼ τ " := (bisimilarity σ τ ).

Definition bisimulation (R : relation (Stream nat)) : Prop :=
  ∀ (σ  τ : Stream nat),
   R σ τ → σ (0) = τ (0) ∧ R σ 'τ '.

Notation block := (list polynomial).

Definition create_horner_block (n x : nat) (cs : polynomial) : block :=
  match n with
    | 0 ⇒ []
    | S n' ⇒ let cs' := horner_poly_div cs x in
             cs' :: create_horner_block_acc n' x cs'
  end.
```

```
Fixpoint create_horner_block_acc (n x : nat) (cs : polynomial) : block :=
  match n with
    | O ⇒ []
    | S n' ⇒ let cs' := removelast (horner_poly_div cs x) in
            cs' :: create_horner_block_acc n' x cs'
  end.

Fixpoint create_triangle_horizontally (xs ys : tuple) : triangle :=
  match ys with
    | [] ⇒ []
    | [y] ⇒ []
    | y :: (_ :: _) as ys' ⇒
      let xs' := make_tuple xs y
      in xs' :: (create_triangle_horizontally xs' ys')
  end.

Fixpoint create_triangle_vertically (xs ys : tuple) : triangle :=
  match xs with
    | [] ⇒ []
    | [x] ⇒ []
    | x :: (_ :: _) as xs' ⇒
      let ys' := make_tuple ys x
      in ys' :: (create_triangle_vertically xs' ys')
  end.

Fixpoint create_triangles_vertically (n : nat) (xs ys : tuple)
  : list triangle :=
  match n with
    | 0 ⇒ [create_triangle_vertically xs ys]
    | S n' ⇒
      let ts := create_triangle_vertically xs ys
      in ts :: (create_triangles_vertically n' xs
                 (rev (cons 0 (hypotenuse ts))))
  end.

CoFixpoint drop (i k : nat) (σ  : Stream nat) : Stream nat :=
  match i with
    | 0 ⇒ (σ ')(0) ::: drop (k - 2) k σ ''
    | S i' ⇒ σ (0) ::: drop i' k σ '
  end.

Inductive Entry : Type :=
  | entry : nat → nat → nat → Entry.

Definition hd {A : Type} (d : A) (xs : list A) : A :=
  match xs with
    | [] ⇒ d
    | x :: _ ⇒ x
  end.
```

```
Definition hd {A : Type} (σ  : Stream A) :=
  match σ with
    |  s ::: _ ⇒ s
  end.

Notation "σ  (0)" := (hd σ ) (at level 8, left associativity).

Definition horner_poly_div (cs : polynomial) (x : nat) : polynomial :=
  match cs with
    | [] ⇒ []
    | c :: cs’ ⇒ c :: (horner_poly_div_acc cs’ x c)
  end.

Fixpoint horner_poly_div_acc (cs’ : polynomial) (x a : nat) :
  polynomial :=
  match cs’ with
    | [] ⇒ []
    | c’ :: cs’’ ⇒
      (c’ + (x ∗ a)) :: (horner_poly_div_acc cs’’ x (c’ + (x ∗ a)))
  end.

Definition horner_poly_eval (cs : polynomial) (x : nat) : nat :=
  horner_poly_eval_acc cs x 0.

Fixpoint horner_poly_eval_acc (cs : polynomial) (x a : nat) : nat :=
  match cs with
    | [] ⇒ a
    | c :: cs’ ⇒ horner_poly_eval_acc cs’ x (c + x ∗ a)
  end.

Fixpoint hypotenuse (ts : triangle) : tuple :=
  match ts with
    | [] ⇒ []
    | t :: ts’ ⇒ (last t 0) :: (hypotenuse ts’)
  end.

Fixpoint last {A : Type} (xs: list A) (d : A) : A :=
  match xs with
    | [] ⇒ d
    | [x] ⇒ x
    | x :: xs ⇒ last xs d
  end.

Fixpoint length {A : Type} (xs : list A) : nat :=
  match xs with
   | [] ⇒ O
   | _ :: xs’ ⇒ S (length xs’)
  end.

Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

```
Notation " [ ] " := nil : list_scope.
Notation " [ x ] " := (cons x nil) : list_scope.
Notation " [ x ; .. ; y ] " := (cons x .. (cons y nil) ..) : list_scope.
Infix "::" := cons (at level 60, right associativity) : list_scope.

Definition list_constant (n c : nat) : list nat :=
  make_list n c (λ x : nat ⇒ x).

Fixpoint list_map (f : nat → nat) (xs : list nat) : list nat :=
  match xs with
    | [] ⇒ xs
    | x :: xs' ⇒ (f x) :: (list_map f xs')
  end.

Definition list_partial_sums (xs : list nat) : list nat :=
  list_partial_sums_acc 0 xs.

Fixpoint list_partial_sums_acc (a : nat) (xs : list nat) : list nat :=
  match xs with
    | [] ⇒ []
    | x :: xs' ⇒ (x + a) :: (list_partial_sums_acc (x + a) xs')
  end.

Definition list_product (xs ys : list nat) : list nat :=
  list_zip mult xs ys.
Infix "⊙ " := list_product (at level 40, left associativity).

Definition list_scalar_multiplication (k : nat)
         (xs : list nat) : list nat :=
  list_map (mult k) xs.
Notation "k ⊗ xs" := (list_scalar_multiplication k xs)
                      (at level 40, left associativity).

Definition list_successor (n i : nat) : list nat :=
  make_list n i S.

Definition list_sum (xs ys : list nat) : list nat :=
  list_zip plus xs ys.
Infix "⊕ " := list_sum (at level 50, left associativity).

Fixpoint list_zip (f : nat → nat → nat)
        (xs ys : list nat) : list nat :=
  match xs, ys with
    | xs, [] ⇒ xs
    | [], ys ⇒ ys
    | x :: xs', y :: ys' ⇒ (f x y) :: (list_zip f xs' ys')
  end.
```

```
CoFixpoint long_stream (n r d c : nat) : Stream nat :=
  (nth 0 (hypotenuse
           (nth n
                (create_triangles_vertically
                   n
                   (tuple_constant (S (S (S r))) 0)
                   ((Str_prefix (S (S (S r)))
                                (0 ::: (p_monomials 0 r 0 c))) ⊕
                   (Str_prefix (S (S (S r)))
                               (p_monomials 0 (S r) 0 d)))) [])) 1)
    ::: (long_stream (S n) r d c).

Fixpoint make_list (n i : nat) (f : nat → nat) : list nat :=
  match n with
    | 0 ⇒ []
    | S n' ⇒ i :: (make_list n' (f i) f)
  end.

CoFixpoint make_stream (f : nat → nat) (n : nat) : Stream nat :=
  n ::: make_stream f (f n).

Fixpoint make_tuple (xs : tuple) (a : nat) : tuple :=
  match xs with
  | [] ⇒ []
  | [x] ⇒ []
  | x :: (_ :: _) as xs' ⇒
    let a' := x + a
    in a' :: make_tuple xs' a'
  end.

CoFixpoint moessner_entries (r n k t : nat) : Stream nat :=
  (moessner_entry r n k t) :::
  (moessner_entries r n (S k) t).

Fixpoint moessner_entry (r n k t : nat) : nat :=
  match n with
    | 0 ⇒ match k with
           | 0 ⇒ 1
           | S k' ⇒ 0
         end
    | S n' ⇒ match k with
              | 0 ⇒ monomial t r (S n') +
                    moessner_entry r n' 0 t
              | S k' ⇒ moessner_entry r n' (S k') t +
                       moessner_entry r n' k' t
            end
  end.
```

```
CoFixpoint moessner_stream (n r : nat) : Stream nat :=
  (nth 0 (hypotenuse
           (nth n
              (create_triangles_vertically
                 n
                 (tuple_constant (S (S r)) 0)
                 (Str_prefix (S (S r)) (monomials 0 r 0)))
              [])) 1)
    ::: (moessner_stream (S n) r).

Definition monomial (t r n : nat) : nat :=
  C(r, n) * (t ^ n).

CoFixpoint monomials (t r n : nat) : Stream nat :=
  (monomial t r n) ::: (monomials t r (S n)).

Fixpoint monomials_list (t r n : nat) : list nat :=
  match n with
    | 0 ⇒ [monomial t r 0]
    | S n' ⇒ (monomial t r (S n')) :: (monomials_list t r n')
  end.

CoFixpoint monomials_sum (t r n a : nat) : Stream nat :=
  let a' := (monomial t r n) + a in
  a' ::: (monomials_sum t r (S n) a').

Fixpoint nth {A : Type} (n : nat) (xs: list A) (d : A) : A :=
  match n, xs with
    | O, x :: xs' ⇒ x
    | O, [] ⇒ d
    | S n', [] ⇒ d
    | S n', x :: xs ⇒ nth n' xs' d
  end.

CoFixpoint p_moessner_stream (n r d : nat) : Stream nat :=
  (nth 0 (hypotenuse
           (nth n
              (create_triangles_vertically
                 n
                 (tuple_constant (S (S r)) 0)
                 (Str_prefix (S (S r)) (p_monomials 0 r 0 d)))
              [])) 1)
    ::: (p_moessner_stream (S n) r d).

Definition p_monomial (x n k d : nat) : nat :=
  d * C(n,k) * x ^ k.

CoFixpoint p_monomials (t r n d : nat) : Stream nat :=
  (p_monomial t r n d) ::: (p_monomials t r (S n) d).
```

```
Inductive Pascal : Entry → Prop :=
| pascal_base_n_0 : ∀ (n : nat), (Pascal (entry n 0 1))
| pascal_base_n_n : ∀ (n : nat), 0 < n → (Pascal (entry n n 1))
| pascal_base_n_lt_k : ∀ (n k : nat), n < k → (Pascal (entry n k 0))
| pascal_inductive_S_n' : ∀ (n' k' v'' v' v : nat),
               v = v'' + v' →
               (Pascal (entry n' k' v'')) →
               (Pascal (entry n' (S k') v')) →
               (Pascal (entry (S n') (S k') v)).

Notation polynomial := (list nat).

Fixpoint removelast {A : Type} (xs : list A) : list A :=
  match xs with
    | [] ⇒ []
    | [x] ⇒ []
    | x :: xs ⇒ x :: removelast xs
  end.

Fixpoint repeat_make_tuple (ys : tuple) (a n : nat) : tuple :=
  match n with
    | 0 ⇒ ys
    | S n' ⇒ repeat_make_tuple (make_tuple ys a) a n'
  end.

Fixpoint rev {A : Type} (xs : list A) : list A :=
  match xs with
    | [] ⇒ []
    | x :: xs' ⇒ rev xs' ++ [x]
  end.

Definition rotated_binomial_coefficient (r c : nat) : nat :=
  C(c + r, c).
Notation "R( r , c )" := (rotated_binomial_coefficient r c).

CoFixpoint rotated_moessner_entries (n r c t : nat) : Stream nat :=
  (rotated_moessner_entry n r c t) :::
  (rotated_moessner_entries n (S r) c t).

Definition rotated_moessner_entry (n r c t : nat) : nat :=
  moessner_entry n (c + r) c t.

Inductive Rotated_Pascal : Entry → Prop :=
| rotated_pascal_base_r_0 :
    ∀ (r : nat), (Rotated_Pascal (entry r 0 1))
| rotated_pascal_base_0_c :
    ∀ (c : nat), 0 < c → (Rotated_Pascal (entry 0 c 1))
| rotated_pascal_induction_r_c :
    ∀ (r' c' v'' v' v : nat),
      v = v'' + v' →
      (Rotated_Pascal (entry (S r') c' v'')) →
      (Rotated_Pascal (entry r' (S c') v')) →
      (Rotated_Pascal (entry (S r') (S c') v)).
```

```
Fixpoint sieve (i k n : nat) (σ  : Stream nat) : Stream nat :=
  match n with
    | 0 ⇒ sieve_step i k σ
    | S n' ⇒ sieve_step i k (sieve (S i) (S k) n' σ )
  end.

Definition sieve_step (i k : nat) (σ  : Stream nat) :=
  stream_partial_sums (drop i k σ ).

CoInductive Stream (A : Type) : Type :=
  Cons : A → Stream A → Stream A.
Notation "s ::: σ " := (Cons s σ ) (at level 60, right associativity).

Definition stream_constant (c : nat) : Stream nat :=
  make_stream (λ x : nat ⇒ x) c.
Notation "# c" := (stream_constant c) (at level 4, left associativity).

CoFixpoint stream_map (f : nat → nat)
          (σ  : Stream nat) : Stream nat :=
  f σ (0) ::: stream_map f σ '.

Definition stream_partial_sums (σ  : Stream nat) : Stream nat :=
  stream_partial_sums_acc 0 σ .

CoFixpoint stream_partial_sums_acc (a : nat)
          (σ  : Stream nat) : Stream nat :=
  σ (0) + a ::: stream_partial_sums_acc (σ (0) + a) σ '.

Definition stream_product (σ  τ : Stream nat) : Stream nat :=
  stream_zip mult σ τ .
Infix "⊙ " := stream_product (at level 40, left associativity).

Definition stream_scalar_multiplication (k : nat)
          (σ  : Stream nat) : Stream nat :=
  stream_map (mult k) σ .
Notation "k ⊗ σ " := (stream_scalar_multiplication k σ )
                    (at level 40, left associativity).

Definition stream_successor (i : nat) : Stream nat :=
  make_stream S i.

Definition stream_sum (σ  τ : Stream nat) : Stream nat :=
  stream_zip plus σ τ .
Infix "⊕ " := stream_sum (at level 50, left associativity).

CoFixpoint stream_zip (f : nat → nat → nat)
          (σ  τ : Stream nat) : Stream nat :=
  f σ (0) τ (0) ::: stream_zip f σ 'τ '.

Definition Str_nth {A : Type} (n : nat) (σ  : Stream A) : A :=
  (Str_nth_tl n σ )(0).
```

```
Fixpoint Str_nth_tl {A : Type} (n : nat) (σ : Stream A) : Stream A :=
  match n with
  | O ⇒ σ
  | S n' ⇒ Str_nth_tl n' σ′
  end.

Fixpoint Str_prefix (n : nat) (σ : Stream nat) : list nat :=
  match n with
    | 0 ⇒ []
    | S n' ⇒ σ (0) :: (Str_prefix n' σ′)
  end.

CoFixpoint successive_powers (b e : nat) : Stream nat :=
  (S b) ^ e ::: successive_powers (S b) e.

Definition tl {A : Type} (xs : list A) : list A :=
  match xs with
    | [] ⇒ []
    | _ :: xs' ⇒ xs'
  end.

Definition tl {A : Type} (σ : Stream A) :=
  match σ with
    | _ ::: σ ' ⇒ σ '
  end.

Notation "σ ′" := (tl σ ) (at level 8, left associativity).

Notation tuple := (list nat).
Notation triangle := (list tuple).

Definition upgrade_seed_tuple (t : nat) (xs : tuple) : tuple :=
  upgrade_seed_tuple_aux t 0 xs.

Fixpoint upgrade_seed_tuple_aux (t a : nat) (xs : tuple) : tuple :=
  match xs with
    | [] ⇒ [a]
    | x :: xs' ⇒ (S t) * x + a :: upgrade_seed_tuple_aux t x xs'
  end.
```

Appendix B

Coq Proofs

```
Corollary binomial_coefficient_eq_rotated_binomial_coefficient :
  ∀ (n k : nat),
    k ≤ n →
    C(n, k) = R(n - k, k).

Lemma binomial_coefficient_implies_Pascal :
  ∀ (n k v : nat),
    v = C(n, k) → Pascal (entry n k v).

Theorem binomial_coefficient_is_symmetric :
  ∀ (n k : nat),
    k ≤ n →
    C(n, k) = C(n, n - k).

Lemma binomial_coefficient_n_eq_k_implies_1 :
  ∀ (n : nat), C(n, n) = 1.

Lemma binomial_coefficient_n_lt_k_implies_0 :
  ∀ (n k : nat), n < k → C(n, k) = 0.

Theorem Binomial_theorem :
  ∀ (t n : nat),
    (S t) ^ n = Str_nth (S n) (stream_partial_sums (monomials t n 0)).

Theorem bisimilarity_iff_Str_nth :
  ∀ (σ  τ : Stream nat),
    σ ∼  τ  ↔  (∀  (n : nat), Str_nth n σ = Str_nth n τ ).

Theorem bisimilarity_iff_Str_nth_tl :
  ∀ (σ  τ : Stream nat),
    σ ∼  τ  ↔  (∀  (n : nat), Str_nth_tl n σ ∼  Str_nth_tl n τ ).

Theorem bisimilarity_iff_Str_prefix :
  ∀ (σ  τ : Stream nat),
    σ ∼  τ  ↔  (∀  (n : nat), Str_prefix n σ = Str_prefix n τ ).
```

```
Lemma bisimilarity_implies_Str_nth :
  ∀ (n : nat) (σ  τ : Stream nat),
    σ ∼  τ → Str_nth n σ = Str_nth n τ .

Lemma bisimilarity_implies_Str_nth_tl :
  ∀ (n : nat) (σ  τ : Stream nat),
    σ ∼  τ → Str_nth_tl n σ ∼  Str_nth_tl n τ .

Lemma bisimilarity_is_a_bisimulation :
  bisimulation bisimilarity.

Theorem bisimilarity_is_reflexive :
  ∀ (σ  : Stream nat),
    σ ∼  σ .

Theorem bisimilarity_is_symmetric :
  ∀ (σ  τ : Stream nat),
    σ ∼  τ →  τ ∼  σ .

Theorem bisimilarity_is_transitive :
  ∀ (σ  τ ρ : Stream nat),
    σ ∼  τ →  τ ∼  ρ →  σ ∼  ρ .

Lemma bisimulation_implies_bisimilarity :
  ∀ (R : relation (Stream nat)),
    bisimulation R  →  ∀ (σ  τ : Stream nat), R σ τ →  σ ∼  τ .

Theorem bisimulation_principle :
  ∀ (σ  τ : Stream nat),
    σ ∼  τ  ↔  ∃ (R : relation (Stream nat)), bisimulation R ∧ R σ τ .

Corollary bottom_element_of_nth_triangle_is_power :
  ∀ (n r : nat),
    nth 0
        (hypotenuse
           (nth n
               (create_triangles_vertically
                  n
                  (tuple_constant (S (S r)) 0)
                  (Str_prefix (S (S r)) (monomials 0 r 0)))
               []))
        1 = (S n) ^ r.
```

```
Corollary bottom_element_of_nth_triangle_is_power_p_monomials_list_sum :
  ∀ (n r d c : nat),
   nth 0
       (hypotenuse
          (nth n
               (create_triangles_vertically
                  n
                  (tuple_constant (S (S (S r))) 0)
                  ((Str_prefix (S (S (S r)))
                               (0 ::: (p_monomials 0 r 0 c))) ⊕
                   (Str_prefix (S (S (S r)))
                               (p_monomials 0 (S r) 0 d)))) [])) 1 =
   (c * (S n) ^ r) + (d * (S n) ^ (S r)).

Theorem correctness_of_repeat_make_tuple :
  ∀ (j : nat) (ys : tuple),
   (repeat_make_tuple ys 0 (S j)) =
   (nth j (create_triangle_vertically
            (tuple_constant (length ys) 0)
            ys)
        []).

Theorem correctness_of_rotated_moessner_entry :
  ∀ (i j r t : nat),
   j ≤ S r →
   S i ≤ S r - j →
   (nth i
        (nth j
             (create_triangle_vertically
                (tuple_constant (S (S r)) 0)
                (Str_prefix (S (S r)) (monomials t r 0)))
             [])
        0) =
   rotated_moessner_entry r i j t.

Theorem correctness_of_upgrade_seed_tuple :
  ∀ (n t : nat),
   upgrade_seed_tuple t (rev (Str_prefix (S n) (monomials (S t) n 0))) =
   rev (Str_prefix (S (S n)) (monomials (S t) (S n) 0)).

Theorem correctness_of_upgrade_seed_tuple_aux :
  ∀ (n t : nat),
   (S t) * (moessner_entry n n 0 t)
            :: upgrade_seed_tuple_aux t (moessner_entry n n 0 t)
            (Str_prefix n (moessner_entries n n 1 t)) =
   Str_prefix (S (S n)) (moessner_entries (S n) (S n) 0 t).

Lemma correctness_of_upgrade_seed_tuple_aux_list :
   ∀ (n r t : nat),
    upgrade_seed_tuple_aux
      t (monomial (S t) r (S n)) (monomials_list (S t) r n) =
    monomials_list (S t) (S r) (S n).
```

```
Theorem create_horner_block_acc_eq_create_triangle_vertically :
  ∀ (r : nat) (σ : Stream nat),
    hypotenuse (create_horner_block_acc
                r 1 (Str_prefix (S r) σ)) =
    hypotenuse (create_triangle_vertically
                (tuple_constant (S r) 0) (Str_prefix (S r) σ)).

Theorem create_horner_block_eq_create_horner_block_acc :
  ∀ (n x : nat) (cs : polynomial),
    create_horner_block n x cs =
    create_horner_block_acc n x (cs ++ [0]).

Corollary create_triangle_vertically_decompose_by_rank :
  ∀ (i j r t : nat),
    j ≤ r → S i ≤ r - j →
    (nth (S i) (nth j
      (create_triangle_vertically
        (tuple_constant (S (S (S r))) 0)
        (Str_prefix (S (S (S r))) (monomials t (S r) 0))) []) 0) =
    t * (nth i (nth j
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0) +
    (nth (S i) (nth j
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0).

Lemma create_triangle_vertically_horizontal_seed_tuple_padding :
  ∀ (r : nat) (σ : Stream nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S r) σ)) =
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant (S r) 0)
          (Str_prefix (S r) σ))) ++ [0].

Corollary create_triangle_vertically_rank_decompose_Pascal_like_c_eq_0 :
  ∀ (i r t : nat),
    i < (S r) →
    (nth (S i) (nth 0
      (create_triangle_vertically
         (tuple_constant (S (S (S r))) 0)
         (Str_prefix (S (S (S r))) (monomials t (S r) 0))) []) 0) =
    (S t) * (nth i (nth 0
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0) +
    monomial t r (S i).
```

```
Corollary create_triangle_vertically_rank_decompose_Pascal_like_c_gt_0 :
  ∀ (i j r t : nat),
    j ≤ r → S i ≤ r - j →
    (nth (S i) (nth (S j)
      (create_triangle_vertically
         (tuple_constant (S (S (S r))) 0)
         (Str_prefix (S (S (S r))) (monomials t (S r) 0))) []) 0) =
    (S t) * (nth i (nth (S j)
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0) +
    (nth (S i) (nth j
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0))) []) 0).

Lemma decompose_Stream :
  ∀ (σ : Stream nat),
    σ = σ (0) ::: σ '.

Lemma equivalence_of_list_partial_sums_acc_and_make_tuple :
  ∀ (xs : tuple) (a : nat),
    make_tuple xs a =
    removelast (list_partial_sums_acc a xs).

Corollary equivalence_of_list_partial_sums_and_make_tuple :
  ∀ (xs : tuple),
    make_tuple xs 0 =
    removelast (list_partial_sums xs).

Theorem equivalence_of_make_tuple_and_list_partial_sums_acc :
  ∀ (xs : tuple) (a : nat),
    make_tuple xs a =
    removelast (list_partial_sums_acc a xs).

Theorem equivalence_of_make_tuple_and_sieve_step_gen :
  ∀ (l i k a : nat) (σ : Stream nat),
    l ≤ i →
    make_tuple (Str_prefix (S l) σ ) a =
    Str_prefix l ((sieve_step i k σ ) ⊕ #a).

Corollary equivalence_of_make_tuple_and_sieve_step :
  ∀ (l : nat) (σ : Stream nat),
    make_tuple (Str_prefix (S l) σ ) 0 =
    Str_prefix l ((sieve_step l (S l) σ )).

Theorem equivalence_of_make_tuple_and_stream_partial_sums_acc :
  ∀ (l' a : nat) (σ : Stream nat),
    make_tuple (Str_prefix (S l') σ ) a =
    Str_prefix l' (stream_partial_sums_acc a σ ).
```

```
Theorem equivalence_of_repeat_make_tuple_and_sieve :
  ∀ (i n : nat) (σ : Stream nat),
    n ≤ i →
    repeat_make_tuple (Str_prefix (S i) σ) 0 (S n) =
    Str_prefix (i - n) (sieve i (S i) n σ).

Corollary equivalence_of_sieve_and_create_triangle_horizontally :
  ∀ (n i : nat) (σ : Stream nat),
    n ≤ i →
    Str_prefix (i - n) (sieve i (S i) n σ) =
    nth n (create_triangle_horizontally
            (Str_prefix (S i) σ)
            (tuple_constant
               (length (Str_prefix (S i) σ)) 0)) [].

Theorem equivalence_of_vertical_and_horizontal_triangle_indices :
  ∀ (i j : nat) (xs ys : tuple),
    (length xs) = (length ys) →
    (nth i (nth j (create_triangle_horizontally xs ys) []) 0) =
    (nth j (nth i (create_triangle_vertically xs ys) []) 0).

Theorem equivalence_of_vertical_and_horizontal_triangle_swap :
  ∀ (xs ys : tuple),
    (create_triangle_horizontally xs ys) =
    (create_triangle_vertically ys xs).

Theorem horner_poly_eval_acc_eq_horner_poly_div_acc :
  ∀ (cs' : polynomial) (c x a : nat),
    horner_poly_eval_acc (c :: cs') x a =
    last (horner_poly_div_acc cs' x (c + x * a)) (c + x * a).

Theorem horner_poly_eval_eq_horner_poly_div :
  ∀ (cs : polynomial) (x : nat),
    horner_poly_eval cs x =
    last (horner_poly_div cs x) 0.

Theorem hypotenuse_create_triangle_vertically :
  ∀ (r t' : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t' r 0))) =
    (rev (Str_prefix (S r) (monomials (S t') r 0))).
```

```
Theorem hypotenuse_create_triangle_vertically_list_sum :
  ∀ (r : nat) (σ  τ : Stream nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant r 0)
         ((Str_prefix r σ ) ⊕
          (Str_prefix r τ ))) =
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant r 0)
          (Str_prefix r σ ))) ⊕
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant r 0)
          (Str_prefix r τ ))).

Theorem hypotenuse_create_triangle_vertically_monomials :
  ∀ (n r t : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S n)) 0)
         (Str_prefix (S (S n)) (monomials t r 0))) =
    (Str_prefix (S n) (moessner_entries r n 0 t)).

Corollary hypotenuse_create_triangle_vertically_p_monomials :
  ∀ (r t d : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (p_monomials t r 0 d))) =
    (rev (Str_prefix (S r) (p_monomials (S t) r 0 d))).

Corollary hypotenuse_create_triangle_vertically_p_monomials_list_sum :
  ∀ (r t d c : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S (S r))) 0)
         ((Str_prefix (S (S (S r))) (0 ::: (p_monomials t r 0 c))) ⊕
          (Str_prefix (S (S (S r))) (p_monomials t (S r) 0 d)))) =
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant (S (S (S r))) 0)
          (Str_prefix (S (S (S r))) (0 ::: (p_monomials t r 0 c))))) ⊕
    (hypotenuse
       (create_triangle_vertically
          (tuple_constant (S (S (S r))) 0)
          (Str_prefix (S (S (S r))) (p_monomials t (S r) 0 d)))).
```

```
Lemma hypotenuse_create_triangle_vertically_remove_padding :
  ∀ (r : nat) (σ  : Stream nat),
    hypotenuse (create_triangle_vertically
                (tuple_constant (S r) 0)
                (Str_prefix (S r) (0 ::: σ ))) =
    hypotenuse (create_triangle_vertically
                (tuple_constant (S r) 0)
                (Str_prefix r σ )).

Theorem hypotenuse_create_triangle_vertically_rotated_moessner_entries :
  ∀ (n r c t : nat),
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S n) 0)
         (Str_prefix (S n) (rotated_moessner_entries r 0 c t))) =
    Str_prefix n (moessner_entries r (c + n) (S c) t).

Lemma length_of_list_sum :
  ∀ (xs ys : list nat),
    length (xs ⊕ ys) = max (length xs) (length ys).

Lemma list_partial_sums_acc_eq_horner_poly_div_acc :
  ∀ (cs : polynomial) (a : nat),
    list_partial_sums_acc a cs =
    horner_poly_div_acc cs 1 a.

Lemma list_partial_sums_eq_horner_poly_div :
  ∀ (cs : polynomial),
    list_partial_sums cs =
    horner_poly_div cs 1.

Theorem Long_s_theorem :
  ∀ (b e d c : nat),
    long_stream b e d c ∼
    (c ⊗ (successive_powers b e)) ⊕ (d ⊗ (successive_powers b (S e))).

Theorem Long_s_weak_theorem :
  ∀ (b e d : nat),
    p_moessner_stream b e d ∼ d ⊗ successive_powers b e.

Corollary make_tuple_monomials_eq_monomials_sum :
  ∀ (l t r n a : nat),
    make_tuple (Str_prefix (S l) (monomials t r n)) a =
    Str_prefix l (monomials_sum t r n a).

Corollary make_tuple_monomials_eq_rotated_moessner_entries :
  ∀ (l' n' t : nat),
    make_tuple (Str_prefix (S l') (monomials t n' 0)) 0 =
    Str_prefix l' (rotated_moessner_entries n' 0 0 t).
```

```
Corollary make_tuple_rotated_moessner_entries :
  ∀ (l' n' c' t : nat),
    make_tuple (Str_prefix
                 (S l') (rotated_moessner_entries n' 0 c' t)) 0 =
    Str_prefix l' (rotated_moessner_entries n' 0 (S c') t).

Theorem moessner_entry_eq_binomial_coefficient :
  ∀ (n k r : nat),
    moessner_entry r n k 0 = C(n, k).

Corollary moessner_entry_eq_monomial :
  ∀ (n k t : nat),
    k ≤ n →
    moessner_entry n n (n - k) t =
    monomial (S t) n k.

Lemma moessner_entry_n_eq_k_implies_1 :
  ∀ (n r t : nat),
    moessner_entry r n n t = 1.

Lemma moessner_entry_n_lt_k_implies_0 :
  ∀ (r n k t : nat),
    n < k →
    moessner_entry r n k t = 0.

Theorem moessner_entry_Pascal_s_rule :
  ∀ (n' r k' t : nat),
    moessner_entry r (S n') (S k') t =
    moessner_entry r n' (S k') t +
    moessner_entry r n' k' t.

Corollary moessner_entry_rank_decompose_by_row :
  ∀ (r c n t : nat),
   moessner_entry (S n) (c + S r) c t =
   t * moessner_entry n (c + r) c t +
   moessner_entry n (c + S r) c t.

Corollary moessner_entry_rank_decompose_Pascal_like_c_eq_0 :
  ∀ (r n t : nat),
    moessner_entry (S n) (S r) 0 t =
    S t * moessner_entry n r 0 t +
    monomial t n (S r).

Corollary moessner_entry_rank_decompose_Pascal_like_c_gt_0 :
  ∀ (r c n t : nat),
   moessner_entry (S n) (S c + S r) (S c) t =
   (S t) * moessner_entry n (S c + r) (S c) t +
   moessner_entry n (c + S r) c t.

Theorem Moessner_s_theorem :
  ∀ (b e : nat),
    moessner_stream b e ∼ successive_powers b e.
```

```
Theorem monomial_decompose_rank :
  ∀ (t r' n' : nat),
    monomial t (S r') (S n') =
    monomial t r' (S n') + t * monomial t r' n'.

Lemma monomial_r_eq_n_implies_power :
  ∀ (t r : nat),
    monomial t r r = t ^ r.

Lemma monomials_list_eq_rev_Str_prefix_monomials :
  ∀ (n r t : nat),
    monomials_list t r n =
    rev (Str_prefix (S n) (monomials t r 0)).

Lemma monomials_list_eq_Str_prefix_moessner_entries :
  ∀ (l n t' : nat),
    l ≤ n →
    Str_prefix (S l) (moessner_entries n n (n - l) t') =
    monomials_list (S t') n l.

Lemma nth_make_tuple_list_sum :
  ∀ (n r i j : nat) (σ  τ : Stream nat),
    nth n (make_tuple (Str_prefix r σ ) i) 0 +
    nth n (make_tuple (Str_prefix r τ ) j) 0 =
    nth n ((make_tuple (Str_prefix r σ ) i) ⊕
          (make_tuple (Str_prefix r τ ) j)) 0.

Lemma nth_rev_eq_last :
  ∀ (xs : list nat) (d : nat),
    nth 0 (rev xs) d =
    last xs d.

Theorem nth_triangle_create_triangles_vertically :
  ∀ (n r t : nat),
    nth n
      (create_triangles_vertically
         n
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (monomials t r 0)))
      [] =
    (create_triangle_vertically
       (tuple_constant (S (S r)) 0)
       (Str_prefix (S (S r)) (monomials (n + t) r 0))).
```

```
Theorem nth_triangle_create_triangles_vertically_p_monomials :
  ∀ (n r t d : nat),
    nth n
      (create_triangles_vertically n
         (tuple_constant (S (S r)) 0)
         (Str_prefix (S (S r)) (p_monomials t r 0 d)))
      [] =
    (create_triangle_vertically
       (tuple_constant (S (S r)) 0)
       (Str_prefix (S (S r)) (p_monomials (n + t) r 0 d))).

Theorem nth_triangle_create_triangles_vertically_p_monomials_list_sum :
  ∀ (n r t d c : nat),
    nth n
      (create_triangles_vertically n
         (tuple_constant (S (S (S r))) 0)
         ((Str_prefix (S (S (S r))) (0 ::: (p_monomials t r 0 c))) ⊕
          (Str_prefix (S (S (S r))) (p_monomials t (S r) 0 d)))) [] =
    create_triangle_vertically
      (tuple_constant (S (S (S r))) 0)
      ((Str_prefix (S (S (S r))) (0 ::: (p_monomials (n + t) r 0 c))) ⊕
       (Str_prefix (S (S (S r))) (p_monomials (n + t) (S r) 0 d))).

Corollary partial_sums_rotated_moessner_entries_bisim_next_column :
  ∀ (n c' t : nat),
    (stream_partial_sums (rotated_moessner_entries n 0 c' t)) ~
    (rotated_moessner_entries n 0 (S c') t).

Theorem Pascal_iff_binomial_coefficient :
  ∀ (n k v : nat),
    Pascal (entry n k v) ↔ v = C(n, k).

Lemma Pascal_implies_binomial_coefficient :
  ∀ (n k v : nat),
    Pascal (entry n k v) → v = C(n, k).

Corollary Pascal_implies_Rotated_Pascal :
  ∀ (n k v : nat),
    Pascal (entry (n + k) k v) →
    Rotated_Pascal (entry n k v).

Theorem Pascal_is_symmetric :
  ∀ (n k v : nat),
    k ≤ n →
    (Pascal (entry n k v) ↔
     Pascal (entry n (n - k) v)).

Theorem Pascal_s_rule' :
  ∀ (n' k' : nat),
    C(S n', S k') = C(n', S k') + C(n', k').
```

```
Lemma removelast_horner_poly_div_acc :
  ∀ (cs : polynomial) (x a : nat),
    removelast (horner_poly_div_acc (cs ++ [0]) x a) =
    horner_poly_div_acc cs x a.

Corollary repeat_make_tuple_monomials_eq_moessner_entries :
  ∀ (c l n t : nat),
    repeat_make_tuple
      (Str_prefix (S c + l) (monomials t n 0)) 0 (S c) =
    Str_prefix l (rotated_moessner_entries n 0 c t).

Lemma repeat_make_tuple_monomials_eq_moessner_entries_general :
  ∀ (k j n t : nat),
    j ≤ k →
    Str_prefix (k - j) (rotated_moessner_entries n 0 j t) =
    repeat_make_tuple (Str_prefix (S k) (monomials t n 0)) 0 (S j).

Theorem repeat_make_tuple_rotated_moessner_entries :
  ∀ (c l n t : nat),
    repeat_make_tuple
      (Str_prefix (c + l) (rotated_moessner_entries n 0 0 t)) 0 c =
    Str_prefix l (rotated_moessner_entries n 0 c t).

Lemma rev_Str_prefix_moessner_entries_eq_monomials :
  ∀ (r t' : nat),
    Str_prefix (S r) (moessner_entries r r 0 t') =
    rev (Str_prefix (S r) (monomials (S t') r 0)).

Lemma rev_Str_prefix_monomials :
  ∀ (l n r t : nat),
    rev (Str_prefix (S l) (monomials t r n)) =
    monomial t r (n + l) :: rev (Str_prefix l (monomials t r n)).

Lemma rotated_binomial_coefficient_implies_Rotated_Pascal :
  ∀ (r c v : nat),
    v = R(r, c) → Rotated_Pascal (entry r c v).

Theorem rotated_binomial_coefficient_is_symmetric :
  ∀ (r c : nat), R(r, c) = R(c, r).

Corollary rotated_moessner_entries_bisim_monomials_sum :
  ∀ (n t : nat),
    monomials_sum t n 0 0 ∼
    rotated_moessner_entries n 0 0 t.

Lemma rotated_moessner_entries_Pascal_s_rule :
  ∀ (n r' c' t : nat),
    rotated_moessner_entries n (S r') (S c') t ∼
    (rotated_moessner_entries n (S r') c' t) ⊕
    (rotated_moessner_entries n r' (S c') t).
```

```
Lemma rotated_moessner_entry_c_eq_0 :
  ∀ (r' n t : nat),
    rotated_moessner_entry n (S r') 0 t =
    monomial t n (S r') + rotated_moessner_entry n r' 0 t.

Lemma rotated_moessner_entry_constrained_negative_Pascal_s_rule :
  ∀ (n' k' t : nat),
    S k' ≤ n' →
    rotated_moessner_entry n' k' (n' - k') t +
    rotated_moessner_entry n' (S k') (n' - S k') t =
    rotated_moessner_entry n' (S k') (n' - k') t.

Theorem rotated_moessner_entry_eq_monomial :
  ∀ (n k t : nat),
    k ≤ n →
    rotated_moessner_entry n k (n - k) t =
    monomial (S t) n k.

Corollary rotated_moessner_entry_eq_rotated_binomial_coefficient :
  ∀ (n r c : nat),
    rotated_moessner_entry n r c 0 = R(r, c).

Lemma rotated_moessner_entry_Pascal_s_rule :
  ∀ (n r' c' t : nat),
    rotated_moessner_entry n (S r') (S c') t =
    rotated_moessner_entry n r' (S c') t +
    rotated_moessner_entry n (S r') c' t.

Lemma rotated_moessner_entry_r_eq_0_implies_1 :
  ∀ (c n t : nat),
    rotated_moessner_entry n 0 c t = 1.

Theorem rotated_moessner_entry_rank_decompose_by_row :
  ∀ (r c n t : nat),
  rotated_moessner_entry (S n) (S r) c t =
  t * rotated_moessner_entry n r c t +
  rotated_moessner_entry n (S r) c t.

Corollary rotated_moessner_entry_rank_decompose_Pascal_like_c_eq_0 :
  ∀ (r n t : nat),
    rotated_moessner_entry (S n) (S r) 0 t =
    (S t) * rotated_moessner_entry n r 0 t +
    monomial t n (S r).

Corollary rotated_moessner_entry_rank_decompose_Pascal_like_c_gt_0 :
  ∀ (r c n t : nat),
  rotated_moessner_entry (S n) (S r) (S c) t =
  (S t) * rotated_moessner_entry n r (S c) t +
  rotated_moessner_entry n (S r) c t.
```

```
Theorem Rotated_Pascal_iff_Pascal :
  ∀ (n k v : nat),
    Rotated_Pascal (entry n k v) ↔
    Pascal (entry (n + k) k v).

Theorem Rotated_Pascal_iff_rotated_binomial_coefficient :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) ↔ v = R(r, c).

Corollary Rotated_Pascal_implies_Pascal :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) →
    Pascal (entry (r + c) c v).

Lemma Rotated_Pascal_implies_rotated_binomial_coefficient :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) → v = R(r, c).

Theorem Rotated_Pascal_is_symmetric :
  ∀ (r c v : nat),
    Rotated_Pascal (entry r c v) ↔
    Rotated_Pascal (entry c r v).

Theorem rotated_Pascal_s_rule' :
  ∀ (r' c' : nat),
    R(S r', S c') = R(S r', c') + R(r', S c').

Lemma shift_make_tuple_create_triangle_vertically :
  ∀ (j' : nat) (ys : tuple),
   make_tuple
     (nth j'
          (create_triangle_vertically
             (tuple_constant (length ys) 0) ys) []) 0 =
   nth (S j')
       (create_triangle_vertically
          (tuple_constant (length ys) 0) ys) [].

Lemma shift_make_tuple_in_repeat_make_tuple :
  ∀ (ys : tuple) (n a : nat),
    make_tuple (repeat_make_tuple ys a n) a =
    repeat_make_tuple (make_tuple ys a) a n.

Lemma shift_start_index_monomials_sum :
  ∀ (i' r n a t : nat),
    (monomial t r n) + (Str_nth i' (monomials_sum t r (S n) a)) =
    (monomial t r (n + (S i'))) + (Str_nth i' (monomials_sum t r n a)).

Lemma stream_partial_sums_acc_monomials_bisim_monomials_sum :
  ∀ (t r n a : nat),
    stream_partial_sums_acc a (monomials t r n) ~
    monomials_sum t r n a.
```

```
Corollary stream_partial_sums_monomials_bisim_monomials_sum :
  ∀ (t r n : nat),
    stream_partial_sums (monomials t r n) ∼
    monomials_sum t r n 0.

Lemma Str_nth_implies_bisimilarity :
  ∀ (σ  τ : Stream nat),
    (∀  (n : nat), Str_nth n σ = Str_nth n τ ) → σ ∼  τ .

Lemma Str_nth_moessner_entries :
  ∀ (i r n k t : nat),
    Str_nth i (moessner_entries r n k t) =
    moessner_entry r n (i + k) t.

Theorem Str_nth_monomials_sum_eq_rotated_moessner_entry :
  ∀ (r t n a : nat),
    Str_nth r (monomials_sum t n 0 a) =
    (rotated_moessner_entry n r 0 t) + a.

Lemma Str_nth_monomials_sum_stream_derivative :
  ∀ (i n t l a : nat),
    Str_nth i (monomials_sum t l n a)′ =
    Str_nth i ((monomials_sum t l (S n) a) ⊕ #(monomial t l n)).

Theorem Str_nth_partial_sums_rotated_moessner_entries :
  ∀ (i n c’ t : nat),
    Str_nth i (stream_partial_sums
                (rotated_moessner_entries n 0 c’ t)) =
    Str_nth i (rotated_moessner_entries n 0 (S c’) t).

Lemma Str_nth_rotated_moessner_entries :
  ∀ (i n r c t : nat),
    Str_nth i (rotated_moessner_entries n r c t) =
    rotated_moessner_entry n (r + i) c t.

Lemma Str_nth_rotated_moessner_entries_over_r :
  ∀ (i n r’ c’ t : nat),
    Str_nth i (rotated_moessner_entries n (S r’) (S c’) t) =
    Str_nth i (stream_partial_sums_acc
                (rotated_moessner_entry n r’ (S c’) t)
                (rotated_moessner_entries n (S r’) c’ t)).

Lemma Str_nth_tl_implies_bisimilarity :
  ∀ (σ  τ : Stream nat),
    (∀  (n : nat), Str_nth_tl n σ ∼ Str_nth_tl n τ ) → σ ∼  τ .

Lemma Str_prefix_drop_gen :
  ∀ (l i k : nat) (σ  : Stream nat),
    l ≤ i →
    Str_prefix l (drop i k σ ) =
    Str_prefix l σ .
```

```
Lemma Str_prefix_sieve :
  ∀ (n l i : nat) (σ : Stream nat),
    l ≤ i →
    Str_prefix l (sieve (S i) (S (S i)) n σ) =
    Str_prefix l (sieve (S (S i)) (S (S (S i))) n σ).

Lemma tl_nth_tl :
  ∀ (n : nat) (σ : Stream nat),
    (Str_nth_tl n σ)′ = Str_nth_tl n σ′.

Lemma unfold_rotated_binomial_coefficient_base_case_0_c :
  ∀ (c : nat),
    0 < c → R(0, c) = 1.

Lemma unfold_rotated_binomial_coefficient_base_case_r_0 :
  ∀ (r : nat),
    R(r, 0) = 1.

Corollary upgrade_seed_tuple_create_triangle_vertically :
  ∀ (n t : nat),
    upgrade_seed_tuple
      t (hypotenuse
           (create_triangle_vertically
              (tuple_constant (S (S n)) 0)
              (Str_prefix (S (S n)) (monomials t n 0)))) =
    hypotenuse
      (create_triangle_vertically
         (tuple_constant (S (S (S n))) 0)
         (Str_prefix (S (S (S n))) (monomials t (S n) 0))).
```