Teaching Divisibility and Binomials with Coq

Sylvie Boldo, François Clément, David Hamelin, Micaela Mayero, Pierre Rousselin

Teaching Divisibility and Binomials with Coq*

Sylvie Boldo[1], François Clément[2], David Hamelin[1], Micaela Mayero[3,1], and Pierre Rousselin[4,2]

[1] Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, LMF, 91190, Gif-sur-Yvette, France.
E-mail: {sylvie.boldo,david.hamelin}@inria.fr
[2] a. Inria, 2 rue Simone Iff, 75589 Paris, France.
b. CERMICS, École des Ponts, 77455 Marne-la-Vallée, France.
E-mail: francois.clement@inria.fr
[3] LIPN, Université Paris 13 - USPN, CNRS UMR 7030, 99 avenue Jean-Baptiste Clément, 93430 Villetaneuse, France.
E-mail: mayero@lipn.univ-paris13.fr
[4] LAGA, Université Paris 13 - USPN, CNRS UMR 7030, 99 avenue Jean-Baptiste Clément, 93430 Villetaneuse, France.
E-mail: rousselin@univ-paris13.fr

Abstract

The goal of this contribution is to provide worksheets in Coq for students to learn about divisibility and binomials. These basic topics are a good case study as they are widely taught in the early academic years (or before in France). We present here our technical and pedagogical choices and the numerous exercises we developed. As expected, it required additional Coq material such as other lemmas and dedicated tactics. The worksheets are freely available and flexible in several ways.

1 Introduction

This work was done in the context of the LiberAbaci project[1] aimed at improving the accessibility of the Coq proof system for mathematics students in the early academic years.

A difficulty in France is that there are many different curricula for the first academic years, so it was difficult to choose which topic to cover. A solution was to focus on just before higher education. At 17–18 years old, French students are usually at the last year of high school (*lycée*), this year being called *terminale* is ending by an exam called *baccalauréat*. With the high school reform from 2020 in France, students may choose mathematics as major (*spécialité*), possibly with additional mathematics during the last year. We have focused on these national curricula, more precisely on integers: divisibility and binomials. It is not the first time Coq tackles mathematics from *baccalauréat*: in 2013, the real analysis exercise was done in Coq at the same time as students [10] as a test of the Coquelicot library [4].

As for the exercises, they follow the spirit from Software Foundations [12] that was also used at Sorbonne Paris-Nord University [9] of a Coq file with holes. We provide exercises with comments both in French and English (as the language should not be the main difficulty for the students). Depending on the difficulty of the exercise, we may provide hints or explanations. Note that we have not yet used these on students.

We provide several levels of leeway to the teacher. First, the teacher may choose among the exercise (and their order) from the worksheets (files named `WS_*.v`). Second, some lemmas of

*This work was supported by the Inria Challenge LiberAbaci.
This work was also partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme – Grant Agreement n°810367.

[1] https://liberabaci.gitlabpages.inria.fr/

the core library can be seen as an exercise, such as the committee-chair identity about binomials in Section 3.2.1. Third, the level of automation is also up to the teacher. We provide all the exercise proofs without automation such as `lia` but it may be allowed for more advanced proofs in order to shorten them. Last but not least, the teacher chooses the environment to be used by the students, both the IDE (possibly jsCoq) and the required modules and libraries (with a limited number of lemmas, with or without automation and so on).

Our goal is that formal proofs be as near as possible to mathematical proofs. Coq is here to prevent the student from making errors, forgetting a sub-case, and so on, as much as the student needs, and without getting on the teacher's nerves. We want the statements and proofs to be as near as possible to the mathematical reasonings and pedagogy. Even if we want to increase the abstraction level of students, we chose not to rely on too much abstraction in these exercises. A first reason is that it is probably too early in the academic years and a second reason is that error messages would be impossible to understand. This is why we rely on plain Peano naturals, and not on an abstract commutative monoid (where we could have defined sums for instance). We therefore only depend upon the Coq standard library, and not on math-comp or Coquelicot (for sums for instance) that abstract too much for our purpose. See also Section 3.1.

Divisibility and binomials are also commonly studied in the first academic year by students in mathematics and the contributions of this article range from *terminale* to the early academic years. Section 2 is about divisibility and Section 3 about binomials.

All the Coq material (and a useful script explained later) is freely available at the following address: `https://liberabaci.gitlabpages.inria.fr/contenus/ThEdu24/ThEdu24.tar.gz` that compiles with Coq versions 8.16 to 8.19. The worksheets are also available (with their proof for the demo) to be tested in a browser thanks to jsCoq [7] at:

`https://liberabaci.gitlabpages.inria.fr/contenus/ThEdu24/div.html`
`https://liberabaci.gitlabpages.inria.fr/contenus/ThEdu24/binom.html`.

2 Divisibility

The divisibility relation on integers and basic arithmetic in general are often taught in high school and at the beginning of mathematically oriented higher education. Without neglecting historical and practical (e.g. in cryptography) aspects, there are also important pedagogical reasons to teach arithmetic. Indeed, even a very basic course in arithmetic will expose students to many proof schemes, potent and nontrivial algorithms, and predicates of very different forms (primality, divisibility) which require the students to design careful rigorous proofs (sadly, maybe for the first time). In particular, the divisibility relation has many pedagogical benefits : working with an existentially quantified predicate, showing the power of symbolic calculus to prove unexpected facts and using one of many different tools (explicit witness, remainder computation, modular arithmetics, prime factor decomposition, and so on) to solve an exercise. It also serves as an example of a non-total order relation, with algebraic compatibility results.

Section 2.1 presents the state of the art and some technical choices, notably on the use of either `N` or `Z`. A point on primality is given in Section 2.2. For binomials, we did not notice any tactics missing as is the case here. For the simplicity of proofs, we defined several dedicated tactics and constructs in Section 2.3, before describing the worksheet in Section 2.4.

2.1 State of the Art and Technical Choices

We chose definitions and basic lemmas from the Coq standard library[4] to back our exercises. It has many benefits, together with some drawbacks (which will be discussed shortly): it is widely

used, does not expose the user to many abstractions and is also used in the series "Software Foundations" by Pierce *et al.* [12].

Another choice to make is whether we work with natural numbers or integers. Natural numbers are easier to work with in some respects, for instance there is no need to consider units in prime number decompositions and there is only one possible choice for Euclidean division. However, they become impracticable whenever subtraction might be involved, for instance in Bezout's lemma. We chose to stick with integers, a last argument being that the standard library offers much more in terms of arithmetics for integer than for natural numbers.

The standard library defines (in Numbers.BinNums) the type `Z` of integers in two steps. First, `positive` (nonzero) natural numbers are defined as non-empty strings of bits, starting with 1. Then, an integer number in `Z` is 0 or a `positive` number with a sign. This means that equality needs not be redefined and that operations are quite fast.

The bulk of the theory of integers in the standard library is actually implementation independent. (Most) lemmas and conventions which are shared with natural numbers are proved in the Numbers.NatInt sub-theory following an axiomatic module-based approach. Lemmas specific to integers are proved (still in an axiomatic way) in the Numbers.Integer.Abstract sub-theory. Finally, these theories are included in the concrete module ZArith.BinInt.Z.

This module-based approach has some benefits (naming consistency, code factoring) as well as downsides: many concrete lemmas have useless hypotheses (it is frustrating, for instance, to have to prove $0 \leq n$ when $n \in \mathbb{N}$). Another source of disappointment is the power operation, which is specified to be of type $\mathtt{t} \to \mathtt{t} \to \mathtt{t}$, with `t` being e.g. Peano's natural numbers (which is fine, then) or integers (which is certainly not mathematically natural and adds some noise in the proofs). In addition, some lemmas are weaker than expected, even with this factorization effort in mind. In practice, we have to provide a thin layer on top of the standard library to remedy these defects. In addition to this general theory of integers, we use the ZArith.Znumtheory module which contains, for instance Gauss' and Bezout's lemmas.

Finally, let us discuss Euclidean division conventions on $\mathbb{Z}$. This is mostly hidden under the rug in usual mathematics courses. Indeed the mathematics *spécialité* national curriculum[2] only mentions Euclidean division of an integer by a positive natural number. In programming languages there are mostly two flavors of Euclidean divisions on integers : "floor division", where the quotient of a by b is the greatest integer smaller or equal to the rational a/b and "trunc division" where it is the value of the ration a/b truncated towards 0. This is discussed at length in the standard library file Numbers.Integer.Abstract.ZDivFloor which also links to the useful reference [5]. We chose the floor division provided by the standard library. It does not have any mathematical advantage over the others, but its notations are less surprising and it is the sole division considered in Znumtheory. In Coq, there is no primitive notion of partial function, so one should decide the result of the Euclidean division by 0. The Coq development team chose wisely $\forall$ `a : Z, a / 0 = 0` and `a mod 0 = 0`, so that the formula $\forall$ `a b : Z, a = b * (a / b) + a mod b` holds even when `b` $= 0$. This actually simplifies greatly many proofs.

2.2 Computation and prime numbers

One of the distinguishing traits of Coq is its orientation towards computation. Most of the operations are actually defined by (terminating) programs which can compute if given explicit arguments (and usually perform as many reduction steps as they can). For instance the `mod` operation on `Z` actually computes the remainder of an Euclidean division, using an efficient algorithm. Then `mod` is proved to satisfy its specification and all the subsequent theory follows.

[2] https://eduscol.education.fr/document/24574/download.

In contrast to computation, `Prop`-valued predicates such as the divisibility relation in `Z` express a property which is (in general) not obtained by computation. In the standard library, it is expressed in a very natural way with an existential quantifier:

```
Definition divide n m = ∃ p, m = n * p.
```

The usual notation for divisibility allows for more concise statements:

```
Infix "|" := divide (at level 0).
```

Some lemmas serve as a bridge between the computational and the non-computational world:

```
Check Z.mod_div : ∀ a b : Z, a mod b = 0 ↔ (b | a).
```

This means that, during the course of an exercise, one can prove explicit divisibility relations simply by computation:

```
Lemma example_div : (31 | 62744).
Proof. apply Z.mod_div; reflexivity. Qed.
```

Here, the `reflexivity` tactic checks that $62744 \bmod 31$ and 0 are *convertible*, in short, in our case, that the result of the computation of the left operand is indeed 0. This kind of translation between properties and computations is at the heart of the math-comp library [8].

While doing some exercises, we noticed that the standard library lacks a function to decide the primality of an integer. The `prime` predicate in Znumtheory is itself quite surprising[3]:

```
Definition prime (p : Z) : Prop := 1 < p ∧ ∀n, 1 ⩽ n < p → Z.gcd n p = 1.
```

In practice, it proved to be as inconvenient as it is pedagogically debatable. Notice the choice to consider negative numbers non-prime which contradicts the usual mathematical conventions (as well as more advanced ideal based definitions of primality). We provide some alternative formulations, as well as a Boolean-valued function `primeb`, which computes:

```
Lemma prime_no_strict_divisor (p : Z) :
  prime p ↔ 1 < p ∧ ∀a, (a | p) → a = −1 ∨ a = 1 ∨ a = p ∨ a = − p.
Lemma prime_no_strict_divisor_below_sqrt (p : Z) :
  prime p ↔ 1 < p ∧ ∀a, 1 < a ⩽ Z.sqrt p → ¬ (a | p).
Lemma primeb_correct (n : Z) : primeb n = true ↔ prime n.
```

Our `primeb` function simply computes the remainders of the division of an integer $p \geq 2$ by $2, 3, ..., \sqrt{p}$, where $\sqrt{p}$ is the integer (floor) square root of p. This way, it becomes possible in exercises to prove well-known primality results by computation:

```
Lemma prime_101 : prime 101.
Proof. apply primeb_correct; reflexivity. Qed.
```

Of course, our algorithm is quite naive and pales in comparison to state-of-the-art methods such as what can be found, for instance, in the coqprime library [13]. It is simple, probably what the students expect at this point, and can compute reasonably fast up to integers of around 8 decimal digits, and does not require to import a large development.

2.3 Tactics

After the ability to compute whether an integer is prime, another gap in the existing tactics is the ability to easily split into various subcases. More precisely, a recurring theme in divisibility

[3] We have modified it slightly for simplification purposes.

exercises is to study the various possible results of a modulo. For example, an integer has only seven possibilities of values for its modulo 7. This is used in the following exercise:

$$\forall n > 0,\ \left(\exists k > 0, \exists q > 0,\ n = k^2 \wedge n = q^3\right) \Rightarrow (n \equiv 0\,[7] \vee n \equiv 1\,[7])\,.$$

It can be proven by considering the modulo 7 remainders of k^2 and q^3. The mathematical proof is then the following table and the fact that only 0 and 1 appear in both lines.

k, q	0	1	2	3	4	5	6
$k^2 \mod 7$	0	1	4	2	2	4	1
$q^3 \mod 7$	0	1	1	6	1	6	6

The Coq formalization of the property is straightforward, but the proof is tedious as we have to split many cases without any help from the tool:

```
Lemma exercise11 (n k q : Z) (h0n : 0 ⩽ n) (h0k : 0 ⩽ k) (h0q : 0 ⩽ q)
    (hk : n = k * k) (hq : n = q * q * q) : n mod 7 = 0 ∨ n mod 7 = 1.
Proof. assert (n mod 7 = 0 ∨ n mod 7 = 1 ∨ n mod 7 = 2 ∨ n mod 7 = 4) as h1.
– rewrite !hk, Z.mul_mod.
  destruct (Z.eq_dec (k mod 7) 0) as [hk0|hk0].
  + rewrite hk0. tauto. (* solves the goal for k mod 7 = 0 *)
  + destruct (Z.eq_dec (k mod 7) 1) as [hk1|hk1].
    * rewrite hk0. tauto. (* solves the goal for k mod 7 = 1 *)
(*... same for 2,3,4,5 and 6: we have k mod 7 ≠ 0, k mod 7 ≠ 1, ... k mod 7 ≠ 6... *)
    * pose proof (mod_pos_bound k 7); nia. (*... but k mod 7 < 7, so the goal is absurd *)
– assert ((n mod 7) = 0 ∨ (n mod 7) = 1 ∨ (n mod 7) = 6) as h2. (* ... same as h1 *)
lia. Qed.
```

To solve this issue, we have added the `study` tactic, that fits the table construction by making the adequate splitting. It considerably shortens the proof:

```
assert (n mod 7 = 0 ∨ n mod 7 = 1 ∨ n mod 7 = 2 ∨ n mod 7 = 4) as h1.
– rewrite !hk, Z.mul_mod.
  study (k mod 7) between 0 and 6 as h2.
  8: pose proof (Z.mod_pos_bound k 7) as bounds; lia.
  all: rewrite ← h2; tauto.
– assert (n mod 7 = 0 ∨ n mod 7 = 1 ∨ n mod 7 = 6) as h2.
  + rewrite !hq, (Z.mul_mod (q * q)), (Z.mul_mod q q).
    study (q mod 7) between 0 and 6 as h2.
    8: pose proof (Z.mod_pos_bound q 7) as bounds; lia.
    all: rewrite ← h2; tauto.
  + lia.
```

Generating a large number of Coq goals may seem hard to handle for students, but it should motivate them to learn selectors such as `all:` or `2,6:` in order to avoid repetition in the proofs.

Finding all solutions This is not a tactic but a definition that aims to stick to mathematical exercise statements. They will often ask to determine the possible numbers satisfying a given property; for instance "*Find all natural numbers such that n+8 is a multiple of n*". Such statements imply that there exists a finite set satisfying the property: it is the student's job to construct it. To express those problems, we added a new definition `findall`, used as follows:

```
Lemma exo15 : findall (λ n ⇒ 0 ⩽ n ∧ n | n + 8).
```

where `findall {E} (P : E → Prop) := { l : list E & Forall P l ∧ ∀(x : E), P x → In x l }`. In other words, the student must provide a list of all the elements satisfying the given property. We use a Sigma-type, which lives in `Type` rather than `exists` which lives in `Prop`: this forces the student to actually construct the object, rather than using a proof by contradiction.

2.4 Worksheet

We were surprised by the variety and number of exercises we found about divisibility. This has led to a large variety in the formalization and pedagogy of the formal counterpart. Some exercises were straightforward to formalize; for instance, $(6 \mid \mathtt{n}) \leftrightarrow (3 \mid \mathtt{n}) \wedge (2 \mid \mathtt{n})$ is stated exactly like we would on paper, and the formal proof uses the same arguments. Some of the exercises become straightforward thanks to additional lemmas and the contents of Section 2.3.

We found out exercises become pedagogical failures when formalized. This is the case of computational exercises such as 17 | 35^228 + 84^501: Coq computes the modulo without the students having to think about a way to do it themselves. We think one such exercise is interesting for outlining computation, but they miss how to cleverly compute $a^b \mod c$.

Many exercises rely on the decimal representation of number, which is out of the scope of the standard library. We therefore developed definitions and lemmas to state results such as the following ones: 91 divides any number written *abcabc*; 2 divides n if and only if 2 divides the last digit of n; 3 divides n if and only of 3 divides the sum of the digits of n. We chose to define our own inductive type `Digits` rather than using a list of digits, so that the student will not have to deal with empty digits list (for the last digit for instance).

3 Binomials

Given natural numbers n and k satisfying $k \leqslant n$, the *binomial coefficient* $\binom{n}{k}$ is the coefficient of the monomial $a^k b^{n-k}$ in the expansion of the binomial $(a+b)$ to the power n, i.e. we have

$$\forall a, b \in \mathbb{R}, \qquad (a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}. \tag{1}$$

The notation $\binom{n}{k}$ usually reads "n choose k". Indeed, the binomial coefficient also represents the number of subsets of k elements of a set of n elements. Note that this second definition extends naturally to the *irregular* case $n < k$ with the value 0, thus making the binomial coefficient a total function over $\mathbb{N}^2$. Note also that this combinatorial aspect is left aside here since the formalization of subsets is itself a nontrivial issue when teaching purposes are at stake.

Binomial coefficients appear in several fields of mathematics such as combinatorics, probability distributions, and series expansion. They can be generalized to real and complex numbers, by using the gamma function instead of the factorial. Their teaching is of great importance; it usually starts in high school, for example for computing probabilities.

There are mainly two ways to define the binomial coefficients. Either the multiplicative way using factorials or products,

$$\forall n, k \in \mathbb{N}, \qquad \binom{n}{k} \stackrel{\text{def.}}{=} \frac{n!}{k!(n-k)!} \quad \text{when } k \leqslant n, \qquad \binom{n}{k} \stackrel{\text{def.}}{=} 0 \quad \text{when } n < k, \tag{2}$$

or its variant where the left clause (when $k \leqslant n$) is replaced with $\binom{n}{k} \stackrel{\text{def.}}{=} \prod_{i=0}^{k-1} \frac{n-i}{k-i}$. Or the

additive way using sums through Pascal's rule,

$$\forall n, k \in \mathbb{N}, \qquad \binom{n+1}{k+1} \stackrel{\text{def.}}{=} \binom{n}{k} + \binom{n}{k+1} \tag{3}$$

together with the initializations $\binom{n}{0} \stackrel{\text{def.}}{=} 1$, and $\binom{0}{k} \stackrel{\text{def.}}{=} 0$ for $k > 0$. Of course, these definitions are equivalent, and choosing one makes the others lemmas. While the second one naturally defines a natural number, the first one a priori lives in the field of rational numbers. Establishing that the division is actually exact needs a demonstration. Note that the expression using factorials is actually valid on $\mathbb{N}^2$, except when $(n, k) = (0, 1)$, provided the use of the Euclidean division and of both conventions $0! \stackrel{\text{def.}}{=} 1$ and $n - k \stackrel{\text{def.}}{=} 0$ when $n < k$.

Many properties of binomial coefficients involve finite summations of natural numbers, which are not always easy to deal with in a proof assistant. Note also that exercises given to students often deal with general purpose properties that may be reused in other proofs.

3.1 State of the Art

As explained above, binomial coefficient are inseparable from factorials and finite summations of natural numbers. Several implementations of these are available through the Coq community.

To start with, the offer provided by the Coq standard library[4] is not satisfactory, for two main reasons. First, the corpus of lemmas is extremely poor: three lemmas about factorials, five about binomial coefficients, and none about finite summations of natural numbers. Second, Pascal's triangle exhibits in (3) an intrinsic integer nature of binomials. Thus, it is not reasonable for teaching purposes to search for assets in the Reals section of the Coq standard library. Indeed, the binomial coefficient is defined as `C` in Reals.Binomial, and the finite sum of endofunctions on natural numbers as `sum_nat_f` in Reals.Rfunctions. Moreover, binomial `C` is defined in $\mathbb{R}$ through the factorial formula (2) and no projection on natural numbers is provided.

As a consequence, several public developments in Coq also provide binomial coefficients, as they actually need them. For instance, we may cite RSA[5] for a proof of correctness of the RSA algorithm by J. C. Almeida and L. Théry in 1999, Bertrand[6] for a proof of Bertrand's postulate by L. Théry in 2002. Both formalize the summation $f(a)+f(a+1)+\ldots+f(a+n)$, and provide the possibility to extract first and last terms, additivity, distributivity and extensionality results, an induction principle, and an index shift. Bertrand adds splitting, monotony and subtractivity results, and some properties about double summation. Both define the binomials using Pascal's triangle (3) and provide basic properties such as Pascal's rule, a property using factorial, and the binomial formula (1). Bertrand adds positiveness, symmetry and monotony results, as well as an upper bound when n is odd and a lower bound when it is even.

We may also cite the FSets[7] library for finite sets over ordered types [6], in which both definitions of the binomials are provided (using Pascal's triangle (3) and using factorials with the Euclidean division (2)), as well as the proof that they agree.

The Coquelicot[8] library [4] defines finite sums in the abstract setting of any abelian monoid. They are formalized as the summation $f(a) + f(a + 1) + \ldots + f(b)$, which is zero, the neutral element of the monoid, when $b < a$. The support includes the extraction of first and last term, an index shift, splitting, additivity and extensionality results.

[4] https://coq.inria.fr/doc/V8.19.1/stdlib/.
[5] https://github.com/coq-contribs/rsa/, see file Binomial.v.
[6] https://github.com/coq-community/bertrand/, see files Summation.v and Binomial.v.
[7] https://github.com/coq-contribs/fsets/, see file PowerSet.v.
[8] https://gitlab.inria.fr/coquelicot/coquelicot/, see file Hierarchy.v.

On the other end, the math-comp[9] library [11] provides comprehensive support for the three features, respectively accessible through the notations `n`!` for `factorial n`, `\sum_(m≤ i<n) f` for an instantiation of `bigop` for the addition of natural numbers, and `'C(n,k)` for `binomial n k`.

To sum up, none seem suitable for our teaching purposes about finite sums of natural numbers, factorials and binomials. The Coq standard library is not competitive on these topics. Contributions such as RSA, Bertrand, or FSets would come with too many out-of-scope lemmas. The formalizations in Coquelicot [4] and math-comp [11] are way too complex. These provide desirable automation for professionals, but makes it challenging to be used with first-years students. We do not want to expose them to a hierarchy of abstract algebraic structures, advanced syntax, or too general definitions.

3.2 Provided Library about Sums, Factorials, and Binomials

We have developed library-like modules Nat_Compl, Nat_Sum, Nat_Factorial and Nat_Binomial, and worksheet-like modules WS_Helper and WS_Binomial. The library modules contain definitions and the associated main properties. The module Nat_Compl is for complementary properties of basic operations on natural numbers. The module WS_Binomial collects proposals for exercises including statements from the library part and specific results that only have a teaching interest. The module WS_Helper aims at listing relevant results about natural numbers for solving the exercises, and at allowing their use without qualification with the `Nat.` prefix. It is the only module required in WS_Binomial.

The library modules are only based on PeanoNat, Compare_dec and Wf_nat from the Coq standard library. We provide definitions that are structurally identical to their usual mathematical counterparts, i.e. with increasingly indexed sums and products. There is no addition of superficial syntax, such as implicit arguments. There is no use of automation through tactics `auto` or `lia`, and only a few occurrences of `easy` and `now`. Thus, almost all proofs are completely explicit, and use plain Coq style. It is straightforward to incorporate automation in an existing proof, but the other way around is quite cumbersome, and is provided. We have also taken a particular care of the naming of the provided lemmas.

3.2.1 Definitions and Main Properties

The module Nat_Sum is dedicated to iterated summations on natural numbers. Given a function $f : \mathbb{N} \to \mathbb{N}$, among various alternatives, we choose a definition that takes the first index a and the number of summands n, and that is structurally equal to $f(a)+f(a+1)+\ldots+f(a+n-1)$. For convenience, we also provide the sum over a range (with both bounds included).

```
Fixpoint sum_n (a n : nat) (f : nat → nat) : nat :=
  match n with 0 ⇒ 0 | S n' ⇒ sum_n a n' f + f (a + n') end.
Definition sum_range a b f := if le_lt_dec a b then sum_n a (b − a + 1) f else 0.
```

Thus, we have `sum_range 0 n f = sum_n 0 (S n)`. We provide extensionality results, extraction of the first and last terms, concatenation, splitting and extraction of any term, weak and strict monotony results, the zero-sum property, additivity and subtractivity, left and right distributivity of the multiplication, and left and right arbitrary index shifts.

The module Nat_Factorial is dedicated to the support for factorials. Given $n \in \mathbb{N}$, we define `fact n` to structurally represent $1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$. The Coq implementation is as follows.

```
Fixpoint fact (n : nat) : nat := match n with 0 ⇒ 1 | S n' ⇒ fact n' * S n' end.
```

[9] https://github.com/math-comp/math-comp/, see files ssrnat.v, bigop.v and binomial.v.

We provide positiveness, non-nullity and division results, weak and strict monotony results, injectivity results, and some divisibility results.

The module Nat_Binomial is dedicated to the support for binomial coefficients. The definition follows the recurrence relation in Pascal's triangle (3). The Coq implementation is as follows.

```
Fixpoint binom (n p : nat) : nat :=
  match n, p with _, 0 ⇒ 1 | 0, _ ⇒ 0 | S n', S p' ⇒ binom n' p' + binom n' p end.
```

We provide Pascal's rule, nullity and positiveness results, weak and strict monotony results, the committee-chair identity and its iterated version, equality with the factorial formula (2), the cancellation identity, rising sums, the hockey stick identity, the binomial formula,Vandermonde's identity, and the formulas for the sum of binomials and of their square.

3.2.2 Worksheet

Note that most general properties provided in Nat_Binomial are commonly considered as good exercises for the training of students. As such, we provide a few hints and/or comments (both in French and in English) for several lemmas on binomials, that can be transmitted to students with statements stripped off their proofs. The proofs in Nat_Binomial are fully detailed, with very limited use of automation. Of course, teachers are free to let students use `lia` or `easy`, in particular if they are sufficiently advanced in the usage of Coq.

Module WS_Helper provides the lists of results from module Arith.PeanoNat.Nat in the Coq standard library that are used in each library modules described above. Thus, teachers may tune the helper module to the exercises they extract from the library modules.

Here is an example of an exercise, with annotations for the students, and solutions for the teacher, with automation (on the right) or without (on the left). Note also that the comments are provided both in French and English, so as not to introduce a potential additional difficulty. Hints for students may be the output of a Coq command, for instance:

```
Check binom_formula.
  ∀ a b n, (a + b) ^ n = sum_range 0 n (λ k ⇒ binom n k * a ^ k * b ^ (n − k))
```

In order to help students, we may want to reduce what the `Search` command gives them, and this is done in the WS_Helper module. For the following lemma, we only need

```
Export Nat (mul_1_r, pow_1_l).
```

And finally the lemma to be shown is stated. We can also imagine a version where the Coq statement is written by the student from the mathematical one: $\forall n, \in \mathbb{N}, \sum_{k=0}^{n} \binom{n}{k} = 2^n$.

```
(*fr Faire la preuve du lemme suivant et remplacer ‘‘Admitted’’ par ‘‘Qed’’. *)
(*en Fill the proof of next lemma and replace ‘‘Admitted’’ by ‘‘Qed’’. *)
Lemma binom_sum n : sum_range 0 n (λ k ⇒ binom n k) = 2 ^ n.
Proof.
(*fr Transformer 2 en une somme. *) (*en Transform 2 into a sum. *)
(*fr Utiliser la formule du binome. *) (*en Use binomial theorem. *)
(*fr Reduire l'egalite de somme a l'egalite des fonctions. *)
(*en Reduce equality on sums to equality on functions. *)
```

```
(* Begin solution for the teacher. *)
(* Without automation. *)
replace 2 with (1+1) by reflexivity.
rewrite binom_formula.
apply sum_range_ext. intros k Hk.
```

```
(* Begin other solution for the teacher. *)
(* With automation. *)
replace 2 with (1+1) by easy.
rewrite binom_formula.
apply sum_range_ext. intros k Hk.
```

```
rewrite !pow_1_l.
rewrite !mul_1_r. reflexivity.
Qed.
(* End solution for the teacher. *)
```

```
rewrite !pow_1_l; lia.
Qed.
(* End other solution for the teacher. *)
```

A script automatically generates the student version with comments in the selected language. It replaces the tagged solution with command `Admitted` so that the file is still compiling. Regarding the automation, there are several possible levels, as exhibited by the previous example. Another question is the place of variables in statements. In Coq, we may write `Lemma name : ∀ n, ...` or `Lemma name n : ...` Both scripts are equivalent, and have their pedagogical advantages and disadvantages that are not discussed here. We have made an arbitrary choice that can be modified by the teacher.

4 Conclusion

We have shown a Coq development about divisibility and binomials. It consists in additional lemmas, dedicated tactics and 43 exercises in two worksheets. The total is more than 250 lemmas including the 43 exercises, and is 2,500 lines of Coq long, including about 1,000 lines of specification and comments. Ideas of exercises either come from textbooks [2, 1] or the internet, mostly `bibmath.net`; a few directly come from a worksheet of a teacher.

This library is free. Teachers may use part or all of it, with several levels of leeway as explained above: which exercises (from the worksheet or even the library), which automation, which environment. This pedagogical freedom is important to adapt the contents to (i) the level of the students and (ii) the goal of the course. We focus on the early academic years or even a year before, but it means a span of four years leading to more or less mature students. The goal may be either purely mathematics, or mainly logic or a trade-off between both. For example, a teacher may want to prove more advanced lemmas with as much automation as possible while another may prevent automation to focus on deduction rules and logical quantifiers.

A difficult point is how this work relates with the Coq standard library, as we need some of its lemmas but would like to replace others. Another difficult point is the level of notation and abstraction allowed. It is tempting to abstract mathematical objects and make the proofs once, but it creates strange hypotheses as seen in Section 2.1. If a student tries to unfold all definitions, both abstraction and notation may make the goal impossible to read and understand.

The scope of this work is only integer properties. But we think we have spanned over many types of exercises, as done in mathematics. We have studied the ones for which we were not satisfied by the proof in order to develop either additional lemmas or tactics to make them reachable. The contribution is therefore both a Coq library and several Coq worksheets, done with a focus on flexibility and a pedagogical insight. A first perspective is of course for this material to be used on real students, be it by ourselves or not. It may help sharpen some of the exercises. Our next step is more advanced mathematics and we plan to develop a worksheet on the Lebesgue integral. This is at the other end of the early academic years as possible (usually done at the end of the bachelor, three years after *terminale*). It deals with advanced concepts such as real numbers, functions, convergence. The mathematics usually rely on implicit axioms such as extensionality or classical logic. We plan to use a Coq library developed for purely research purpose [3] and see how it can be adapted to obtain pedagogical contents.

Acknowledgments

We are thankful to Mr. Delabrouille for letting us use his paper worksheet exercises for divisibility.

References

[1] *Hyperbole Terminale - Option Mathématiques Expertes*. Nathan, 2020.

[2] *Hyperbole Terminale - Spécialité Mathématiques*. Nathan, 2020.

[3] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq Formalization of Lebesgue Integration of Nonnegative Functions. *Journal of Automated Reasoning*, 66:175–213, 2022. https://inria.hal.science/hal-03471095/.

[4] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. https://inria.hal.science/hal-00860648/.

[5] Raymond T. Boute. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):127–144, 1992. https://doi.org/10.1145/128861.128862.

[6] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In David Schmidt, editor, *Proc. of the 13th European Symposium on Programming (ESOP 2004)*, volume 2986 of *LNCS*, pages 370–384. Springer, Berlin - Heidelberg, 2004. https://doi.org/10.1007/978-3-540-24725-8_26.

[7] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards Hybrid Theorem Proving Interfaces. In S. Autexier and P. Quaresma, editors, *Proc. of the 12th Workshop on User Interfaces for Theorem Provers (UITP 2016)*, volume 239 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–27. Open Publishing Association, 2017. https://doi.org/10.4204/EPTCS.239.2.

[8] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq System. Research Report RR-6455, Inria Saclay - Île-de-France, 2016. https://inria.hal.science/inria-00258384/.

[9] Marie. Kerjean, Frédéric Le Roux, Patrick Massot, Micaela Mayero, Zoé Mesnil, Simon Modeste, Julien Narboux, and Pierre Rousselin. Utilisation des assistants de preuves pour l'enseignement en L1. *Gazette de la Société Mathématique de France*, 174, octobre 2022. https://smf.emath.fr/publications/la-gazette-de-la-societe-mathematique-de-france-174-octobre-2022.

[10] Catherine Lelay. *Repenser la bibliothèque réelle de Coq : vers une formalisation de l'analyse classique mieux adaptée*. Thèse, Université Paris Sud - Paris XI, June 2015. https://theses.hal.science/tel-01228517.

[11] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2022. https://doi.org/10.5281/zenodo.7118596.

[12] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software Foundations. Electronic textbooks, 2023. https://softwarefoundations.cis.upenn.edu.

[13] Laurent Théry and Guillaume Hanrot. Primality proving with elliptic curves. In Klaus Schneider and Jens Brandt, editors, *Proc. of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOL 2007)*, volume 4732 of *LNCS*, pages 319–333. Springer, Berlin - Heidelberg, 2007. https://doi.org/10.1007/978-3-540-74591-4_24.