Introduction to Formal Methods Using Interactive Proof Assistant Rocq

Evgeny M. Makarov

***Abstract*—This article describes our experience of introducing undergraduate students to formal methods using the interactive proof assistant Rocq. The goals of the course include sharpening students' skills in writing strict mathematical proofs both on paper and on a computer, as well as demonstrating practical applications of mathematical logic. At the end of the course students do group projects where they specify and verify an algorithm, such as finding maximum in a one- or two-dimensional array, checking if a number is prime or computing an integer root of an equation.**

We tried to minimize nontrivial aspects of Rocq and use the most of the material already familiar to students. This comparative simplicity, which is appropriate for the first introduction to formal methods, distinguishes the course from its analogs.

***Keywords*—Interactive proof assistant Rocq, Formal methods, Program verification, Education.**

I. Introduction

This article describes the experience of introducing students to formal methods using the interactive proof assistant Rocq. The course is called “Foundations of Computer Science” [1]; it is taught to fourth year undergraduate students majoring in mathematics in the Institute of Information Technologies, Mathematics and Mechanics in Lobachevsky State University of Nizhny Novgorod, Russia.

Formal methods are mathematically based languages, techniques and tools for specifying, developing and verifying software and hardware systems [2]. These tools become indispensable as both system complexity and the price of error grow. That's why formal methods are extensively used in such areas as aerospace industry [3] and design of processors [4] and network protocols [5]. Many tasks, such as program verification, are already achievable at least in theory, though in practice the development of fully verified software is often not feasible due to high cost. Nevertheless, there is hope that rapid development will bring the cost of verification down and formal methods will gain wide acceptance. One evidence of that is several hundred conferences and workshops related to formal methods that take place every year.

Some of the most useful tools employed in formal methods are automated or fully automatic proof assistants, or theorem provers [6]. These programs can be used to formalize both proofs in pure mathematics and proofs related to algorithm correctness. First theorem provers appeared in the 1950s, and the 1980s witnessed the beginning of the development of such systems as Rocq, HOL Light and ACL2, which are still widely used today.

The course described here is not one of the core disciplines in the mathematics curriculum. There are weekly labs that take one and a half hours each. The course covers different topics during the two years that it is taught. Two semesters of the third year support the numerical methods course, while the fall and spring semesters of the last year are devoted to studying functional programming and computer-assisted theorem proving, respectively. The last part of the spring semester is allotted to finishing bachelor's thesis, so this semester lasts for twelve weeks instead of the usual sixteen. This makes it difficult to thoroughly introduce formal methods or even teach working with one proof assistant. Therefore, the course covers only elementary program specification and verification. This is its main difference from other courses, which cover mathematical foundations and subtleties of working with Rocq to a greater degree. A comparison with similar courses is found in Sect. VI.

The greater part of the course is spent learning how to use Rocq, and the last three or four weeks are devoted to group projects. We avoid teaching complex inductive types and predicates and only cover introduction to type theory, which is the foundation of Rocq.

Having studied mathematical logic during the previous semester, students are familiar with first-order formulas, free and bound variables and either natural deduction or Gentzen's sequent calculus. If time allows, the logic course also covers lambda calculus, which underlies type theory used in Rocq. Lambda calculus is also studied in the functional programming course that runs parallel to the logic course. These things are helpful, but not indispensable to studying proof assistants.

Computer-aided software verification, in turn, provides a much-desired real-world application of mathematical logic. Students get a chance not just to write a program, which they have done numerous times in other courses, but to prove its correctness as well. Of course, algorithms offered in course projects are very simple, and it may look like efforts required to explain seemingly obvious details to a computer are not worth it. Some of these difficulties indeed come from the annoying failure of a computer to understand things that are obvious to people; however, others point to errors in the algorithm or its specification. Therefore the level of precision required in this course exceeds the corresponding level in other areas, such as epsilon-delta proofs in calculus. In our opinion, writing such proofs is beneficial to mathematics majors because it demonstrates that no proof step has to rely on intuition; rather, every step is described by precise inference rules.

Manuscript received August 15, 2025.
E. M. Makarov is with the Institute of Information Technologies, Mathematics and Mechanics, Lobachevsky State University, Nizhny Novgorod 603022, Russia (phone: +7-952-777-2045; e-mail: evgeny.makarov@itmm.unn.ru).

The contributions of the article are as follows. The course is described in sufficient detail for anyone familiar with Rocq to implement a similar one. We show that discussing type theory and more advanced aspects of Rocq can be kept to a minimum, and thus it is possible to introduce students to a modern proof assistant even during an abbreviated semester. Finally, we discuss how writing computer-assisted proofs develops skills that are different from those in other mathematical disciplines.

The rest of the article has the following structure. The next section introduces the Rocq proof assistant. Sect. III describes topics studied during the main part of the course. Sect. IV lists examples of projects. Sect. V discusses proof-writing skills. Sect. VI compares this course to similar ones. Finally, Sect. VII sums up and describes future work.

II. Introduction to Rocq Theorem Prover

Interactive theorem prover Coq [7]–[8] has been developed in the French research institute INRIA since the 1980s. In 2025 it was renamed into Rocq. The development is ongoing, and new versions come out approximately every five months. Rocq is one of the most advanced and widely used computer proof systems both in academia and in the industry. This is why working with Rocq is an excellent way of introducing students to formal methods.

Rocq was used to prove a number of nontrivial results both in pure mathematics and in computer science. In the end of the 1990s Rocq was used to formalize a constructive proof of the fundamental theorem of algebra. This theoretically allowed computing a root of every non-constant polynomial with required precision. In practice the program extracted from the proof was too slow. Therefore a library of constructive algebra and analysis was developed that used a more efficient representation of real numbers. The work in this direction is ongoing [9].

In 2005 a complete proof of the four color theorem was formalized, including the correctness of the program that verified a large number of special cases [10]. In the same year Rocq was used to prove the first Gödel's incompleteness theorem [11]. In 2012 the Feit-Thompson Odd Order Theorem was proved in Rocq [12] (the original proof published in 1963 contained 255 pages). An example of an important project in computer science is CompCert—a compiler for a significant fragment of the C programming language whose specification, implementation and verification were all done in Rocq [13]. The complexity of the correctness proof for a program of such scale defies imagination.

Rocq is based on type theory called the Calculus of Inductive Constructions (CIC). It is an expressive variant of typed lambda calculus, which, due to Curry-Howard correspondence, serves both as a functional programming language and as a logical calculus. The main judgment in type theory is $\Gamma \vdash t : A$, which says that in context Γ that contains types of free variables term t has type A. This judgment can also be read as saying that t is a proof of theorem A.

CIC uses dependent types that can contain terms. The most important inference rules are the following.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\texttt{fun}\, x : A \Rightarrow t) : (\forall x : A,\, B)} \qquad \frac{\Gamma \vdash t_1 : (\forall x : A,\, B) \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B[t_2/x]} \tag{1}$$

Here fun x: $A \Rightarrow t$ denotes a function with argument x of type A and body t, and $B[t_2/x]$ denotes the result of substituting term t_2 for variable x in type B. These rules show that $\forall x$: A, B is a dependent product, or the type of functions whose range depends on the value of their argument x. Simultaneously $\forall x$: A, B is a proposition saying that B is true for every x of type A.

Implication $A \to B$, which is also a type of simple functions from A to B, is a contraction of $\forall x$: A, B when x does not occur freely in B. Other logical connectives—conjunction, disjunction and existential quantifier—are defined using inductive types. However, a Rocq user does not need to know this since the Rocq library contains the usual introduction and elimination rules for these connectives.

Another important inference rule of Rocq is the conversion rule, which makes sense due to the presence of dependent types.

$$\frac{\Gamma \vdash t : A \quad A =_{\beta\iota\delta\zeta} B}{\Gamma \vdash t : B}$$

Here β, ι, δ and ζ are different types of reductions on terms, of which the best known is β-reduction.

$$(\text{fun } x : A \Rightarrow t_1)t_2 \to_\beta t_1[t_2/x]$$

Notation $A =_{\beta\iota\delta\zeta} B$ means that type B can be obtained from A using a chain of reduction and reverse reductions. Reductions define the operational semantics of Rocq, and the equivalence relation that they generate is sometimes called *equality by definition*. Thus the rule above says that a proof of a proposition is simultaneously a proof of another proposition that is obtained from the first one by replacing some terms with others that are equal by definition. For example, a constant `eq_refl` has the type, or, equivalently, is a proof of reflexivity of equality $\forall A$: `Type`, $\forall x$: A, $x = x$, so `eq_refl` 4 is a proof of $4 = 4$.[1] However, term $2 + 2$ reduces to 4, so `eq_refl` 4 is also a proof of $2 + 2 = 4$. The distinction between definitional equality and the so-called Leibniz equality, which is most commonly used in mathematics, is described by the Poincaré principle, which postulates the difference between a proof and a verification: the latter is considered trivial and is not considered a part of a proof. The conversion rule allows Rocq, unlike many other proof assistants, using proofs by reflection, when a procedure for deciding a certain problem is both written and verified in Rocq.

Inference rules of CIC are relatively few and well studied in mathematics. Their implementation is collected in a so-called kernel. Other parts of the proof assistant can try to build a proof, but only the kernel checks if the proof is correct. Having a relatively small kernel that can be checked independently is called the de Bruijn principle. It increases trust in the system.

Rocq has natural syntax that makes it easier to use it in education. For example, here is the definition of prime

[1] We omitted the implicit argument `nat` of `eq_refl` which is usually not printed by Rocq.

numbers.

```
Definition prime (n : nat) : Prop :=
  1 < n /\
  forall d, 1 < d -> d < n -> ~(d | n).
```

Rocq also has rich facilities for defining new notations and can use Unicode symbols, which allows, for example, writing `∀ n` instead of `forall n`.

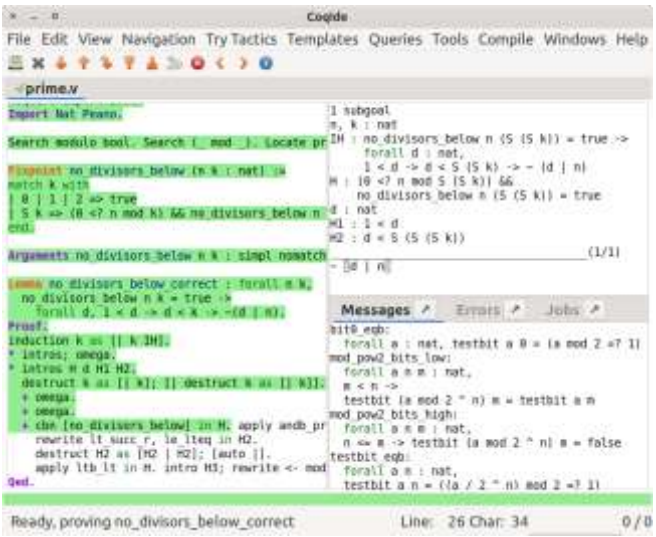

Fig. 1. Integrated development environment RocqIDE

Rocq is freely distributed for Linux, Windows and macOS. It includes an integrated environment RocqIDE for writing proofs. The program window, shown in Fig. 1, consists of three parts. The left one contains a file with definitions, theorems and proofs. A proof consists of so-called tactics, which are admissible inference rules from the logical standpoint. Rocq checks application of each tactic, and when there are no errors, it paints commands that have already been executed in green. This area can no longer be changed by the user, though it is possible to undo several steps or start the proof from the beginning.

The top right part of the window shows the current goal, which is a judgment of the form $\Gamma \vdash t: A$. The context Γ, which contains variables and assumptions used in the proof, is written vertically. Instead of the symbol $\vdash$ there is a horizontal line. Below it is the statement currently being proved. At each moment there can be several goals, and the user works with one of them. Finally, the bottom right corner shows messages, such as results of searching lemmas in the standard library or error messages.

The user starts a proof from the theorem statement and uses tactics to convert it to other goals. Thus, a proof is usually built bottom up, i.e., from the final proposition to axioms. However, there are tactics that produce corollaries of existing theorems and assumptions.

III. Covered Topics

As was said in the introduction, this course is abbreviated and lasts only twelve weeks. There are no dedicated lectures, only labs in a computer class. In the first part of the class the instructor explains the new material, and in the second part students work on assignments. Unsolved exercises are left as homework. Below are topic for each week's classes.

Week 1. Basic syntax. Propositional logic. Connection between Rocq tactics and inference rules from the mathematical logic course. Focusing proofs using special symbols `*`, `+` and `-` to indicate where the proof of each subgoal starts and ends. Simple automatic tactics `trivial`, `auto`, `easy`). Using the integrated environment.

Examples of statements that students should be able to prove.

```
Theorem disj_premise :
  (A \/ B -> C) -> (A -> C) /\ (B -> C).
Theorem deMorgan :
  (~A \/ ~B) -> ~(A /\ B).
```

Week 2. Predicate logic. Tactics working with quantifiers and their comparison with standard logical inference rules. The simplest `;` for joining tactics.

Examples of statements.

```
Theorem t1 : (forall x : T, P x) \/
  (forall x : T, Q x) →
    (forall x : T, P x \/ Q x).
Theorem t2 :
  ((exists x, P x) -> forall x, Q x) →
    forall x, P x -> Q x.
```

Week 3. Leibniz, or provable, equality. Rewriting tactics. Peano arithmetic. Defining functions using recursion; definitional equality that they generate. Proof by induction on natural numbers.

Examples of statements: associativity, commutativity of addition and multiplication; distributivity of multiplication with respect to addition.

Week 4. Automatic tactics for proving statements in Presburger arithmetic and polynomial equalities (`lia` and `ring`). Searching the standard library for theorems. Commands for printing information about terms. Unification of a goal and a statement done by the `apply` tactic.

Examples of statements.

```
Theorem sum_progression :
  forall n, 2 * sum n = n * (n + 1).
Theorem sum_cubes :
  forall n, sumCubes n = (sum n)^2.
```

Week 5. Introduction to Curry-Howard correspondence between terms and proofs and between types and theorems. Simply typed lambda calculus. Dependent types. Explanation why applying a lemma to an argument corresponds to universal quantifier elimination inference rule (1). Proving arithmetic facts.

Example of statements.

```
Theorem lt_S_r :
  forall m n, m < S n -> m = n \/ m < n.
```

Here `S` denotes the successor function that adds 1 to its argument. It is used in Peano axioms.

Week 6. Difference between types `Prop` and `bool`. Using predicates that return `bool` in programs. Proofs by cases on the the truth value of `bool` predicates using reflection of sort `Prop` in type `bool` [14, vol. 1]. Representing an array as a function on natural numbers and a natural number: array's length. Defining recursive functions on arrays, such as finding array's maximum. Full specification of such

functions.

Examples of statements.

```
Theorem arrayMaxSpec1 :
  forall n i,
    i <= n -> array i <= arrayMax n.
Theorem arrayMaxSpec2 :
  forall n, exists i,
    i <= n /\ array i = arrayMax n.
```

Week 7. Finishing verification of the maximum function on arrays. Dependent sums and existential quantification. Program extraction from constructive proofs.

Week 8. Induction with several bases and parameters. Strong induction. Simplifying recursive functions tactics `cbn` and `simpl`). Additional tactics (`discriminate`, `simplify_eq`, `contradict`, `intuition`). Choosing group projects.

Weeks 9–12. Working on projects and presenting them to the class or the instructor.

IV. Course Projects

During the last three or four weeks of the course students work on projects in groups of two or three. In each case the task is to write an algorithm and prove that it conforms to its specification. Unfortunately, there is not enough time to listen to students' presentations, even though creating a presentation and addressing the audience would be a valuable experience.

Project 1. Check if an array is sorted. One has to write a definition and prove a theorem as follows.

```
Fixpoint arraySorted (n: nat): bool := …
Definition sorted (n : nat) : Prop :=
  forall i1 i2, i1 < i2 -> i2 < n →
    array i1 <= array i2.
Theorem arraySortedSpec:
  forall n,
    sorted n <-> arraySorted n = true.
```

Function `arraySorted` checks if an array of length *n* is sorted in non-decreasing order (the array itself is a parameter), and predicate `sorted` is its specification. The theorem relates the two concepts.

Project 2. For given `f : nat → nat` and `n : nat` determine if the restriction of `f` on `{0, ..., n}` is an injection.

Project 3. For given `f : nat → nat` and m `n : nat` determine if `{0, ..., n} ⊆ {f 0, ..., f m}`. In other words, if `f` maps `{0, ..., m}` to `{0, ..., n}`, one must determine if `f` is a surjection.

Project 4. Determine if a number occurs in a 2D array represented by a function of type `nat -> nat -> nat` together with the numbers of rows and columns.

Project 5. Find the greatest divisor (not necessarily prime) of a number smaller than the number itself.

Project 6. Find the greatest common divisor of numbers `m` and `n` using the Euclidean algorithms or the search from `min m n` to 1.

Project 7. Check if a natural number is prime.

Project 8. Check if an array is a palindrome, i.e., `∀ i j, i + j = n → a i = a j` holds.

Project 9. Prove the equivalence of two function definitions that use tail and nontail recursion. One can consider addition, multiplication, factorial and Fibonacci numbers.

In order to prove the equivalence of the highly inefficient but standard function `fib` and the efficient but less obvious function `fibIter` defined as follows:

```
Fixpoint fib (n : nat) :=
match n with
| 0 => 0
| S 0 => 1
| S (S p as q) => fib q + fib p
end.

Fixpoint fibIter (n : nat) (a : nat) (b :
nat) :=
match n with
| 0 => a
| S p => fibIter p b (a+b)
end.
```

one has to come up with a nontrivial invariant, e.g.,

```
Lemma fib_eq_iter :
  forall a n, fib (n+a) =
    fibIter n (fib a) (fib (a+1)).
```

Then the desired result

```
Theorem fib_eq :
  forall n, fibIter n 0 1 = fib n.
```

easily follows.

Project 10. Find an integer root of an equation `f x = y` where `f` is an unbounded function and `f 0 <= y`.

Project 11. Implement addition of two numbers represented by function of type `nat -> nat` and prove its correctness.

Project 12. Find the value of a polynomial at a point using Horner's method and prove that it is equal to the value computed in the usual way.

Project 13. Prove the following theorem.

```
Variables f g : nat -> nat.
Hypothesis notSurjection :
  forall n, g n <> 0.
Theorem fgf : exists n, f (g (f n)) <> n.
```

This statement has a constructive and a nonconstructive proofs. Finding the required *n* without an unbounded search requires some ingenuity.

Project 14. Prove the pigeonhole principle: if `f : {0, …, n} → {0, …, n-1}`, then there exist `i` and `j` such that `0 ≤ i < j ≤ n` and `f(i) = f(j)`.

Functions that modify arrays, such as various sorting algorithms, are also good candidates for course projects, but their proofs are more complicated because one has to show that changing one array element does not affect others.

V. Skills Developed by Writing Formal Proofs

This section describes how working with a proof assistant can help mathematics majors develop proof-writing skills. Prior to this course students have already covered almost all undergraduate curriculum including courses like “Differential Geometry and Topology,” “Number Theory,” “Mathematical Logic” and “Lie Groups and Algebras,” so

the students' ability to comprehend and write complex proofs is in no doubt. This course goes in the direction that is in some sense opposite to the disciplines mentioned above. Indeed, though it covers elementary facts about natural numbers and proofs by induction, attempting to explain a proof to a computer leads to numerous difficulties with statements that are intuitively obvious. The course strives to overcome those difficulties by teaching students construct highly detailed and rigorous proofs and think about proof steps in terms of known inference rules.

This distinction reminds two ways of studying mathematics described by the philosopher and mathematician Bertrand Russell in his book "Introduction to Mathematical Philosophy" [15].

> Mathematics is a study which, when we start from its most familiar portions, may be pursued in either of two opposite directions. The more familiar direction is constructive, towards gradually increasing complexity: from integers to fractions, real numbers, complex numbers; from addition and multiplication to differentiation and integration, and on to higher mathematics. The other direction, which is less familiar, proceeds, by analysing, to greater and greater abstractness and logical simplicity…
>
> We may state the same distinction in another way. The most obvious and easy things in mathematics are not those that come logically at the beginning; they are things that, from the point of view of logical deduction, come somewhere in the middle. Just as the easiest bodies to see are those that are neither very near nor very far, neither very small nor very great, so the easiest conceptions to grasp are those that are neither very complex nor very simple (using "simple" in a logical sense). And as we need two sorts of instruments, the telescope and the microscope, for the enlargement of our visual powers, so we need two sorts of instruments for the enlargement of our logical powers, one to take us forward to the higher mathematics, the other to take us backward to the logical foundations of the things that we are inclined to take for granted in mathematics.

Here are some examples of complications faced by students during proofs of seemingly obvious facts.

Greatest Divisor. Project 5 from Sect. IV asks to write a function $f(n)$ that computes the greatest divisor of n smaller than n itself. One can define an auxiliary function $g(n, k)$ by recursion on k that returns the greatest divisor of n smaller than k; then $f(n) = g(n, n)$. Specification of g says, in particular,

$$\forall n \forall k \forall j\ (g(n, k) < j \rightarrow j < k \rightarrow \neg(j \mid n)) \qquad (2)$$

where $j \mid n$ means that j divides n. This statement is proved by induction on k. In the induction step one has to show that (2) and $g(n, S(k)) < j < S(k)$ imply $\neg(j \mid n)$ (to remind, S is the successor function). If k divides n, then $g(n, S(k)) = k$ and $k < j < S(k)$ is impossible on natural numbers. Otherwise by definition $g(n, S(k)) = g(n, k)$, and one has to show that $g(n, k) < j < S(k)$ implies $\neg(j \mid n)$. This claim is very similar to (2), but if one tries to derive $\neg(j \mid n)$ using (2) at this point, one has to prove its premise $j < k$ from $j < S(k)$, which is impossible. Of course, $\neg(k \mid n)$ holds by assumption, and $\neg(j \mid n)$ for $j < k$ follows from the induction hypothesis.

A large part of difficulties faced by students are similar. Without a clear plan proof attempts can degenerate into a syntactic game where a students tries to apply all available theorems and assumptions. This is why it is important to have a detailed paper proof, which is then consistently being implemented in Rocq. At each step the student must be aware of the state of the formalization process.

Induction with Initial Value Different from Zero. When dealing with prime numbers one often has to prove statements like $\forall n\ (1 < n \rightarrow P(n))$ from $P(2)$ and

$$\forall n(1 < n \rightarrow P(n) \rightarrow P(S(n))). \qquad (3)$$

This has to be done using regular mathematical induction. The base is $1 < 0 \rightarrow P(0)$, which is trivially true. The induction step amounts to showing that $\forall n\ (1 < n \rightarrow P(n))$ implies $\forall n\ (1 < S(n) \rightarrow P(S(n)))$.

An attempt to use (3) to prove $P(S(n))$ leads to the need to prove $1 < n$ from $1 < S(n)$, i.e., $0 < n$, which is impossible. The right approach is to consider cases $n = 0$, $n = 1$ and $1 < n$. In the first case the premise $1 < S(n)$ is false, in the second one $P(S(n))$ is $P(2)$, which holds by assumption, and in the last case $P(S(n))$ follows from the induction hypothesis and (3).

A math major has to know that different types of induction on natural numbers, including strong induction, are derived from the standard induction principle.

Multiparameter Induction. Often the statement proved for all n by induction has the form $\forall k\ P(n, k)$. It is a good idea to keep the quantifier on k in the induction hypothesis instead of fixing k in the whole proof.

As an example, consider decidability of equality on natural numbers, which can be implemented as follows.

```
Fixpoint eqb (x y : nat) : bool :=
match x, y with
| 0, 0 ⇒ true
| 0, S _ ⇒ false
| S _, 0 ⇒ false
| S x1, S y1 ⇒ eqb x1 y1
end.
```

One has to prove $\forall x \forall y\ \texttt{eqb}(x, y) = \texttt{true} \leftrightarrow x = y$). The proof proceeds by induction on x and considering cases $y = 0$ and $y = S(z)$ for some z (the induction hypothesis on y is not needed).

The induction predicate on x should be $\forall y\ (\texttt{eqb}(x, y) = \texttt{true} \leftrightarrow x = y)$ and not just $\texttt{eqb}(x, y) = \texttt{true} \leftrightarrow x = y$ because the claim for $S(x)$ and $S(y)$ are derived from the similar claim for x and y and not for x and $S(y)$. This is an important detail that is easy to miss in paper proofs.

"Without Loss of Generality." Let $P(x, y)$ be a symmetric relation on natural numbers. It is clear that to show $P(x, y)$ for all x and y it is sufficient to consider the case $x \le y$. Usually this is conveyed by the phrase, "Without loss of generality, assume that $x \le y$." However, representing this phrase as a precise inference rule is a nice exercise. In this case $P(x, y)$ is derived from assumptions $\forall u \forall v\ (u \le v \rightarrow P(u, v)) \rightarrow P(x, y)$ and $x \le y \rightarrow P(x, y)$ (see the tactic `wlog` in the description of SSReflect in [8]).

Pigeonhole Principle. This principle, whose statement is given in project 14, is a striking example of a proposition

whose proof contains details one does not think about at first. In fact, a formal proof contains a program that, given a function f with the given domain and codomain, returns distinct numbers i and j such that $f(i) = f(j)$. And yet at first glance this program seems less obvious than the proof that contains it.

The proof proceeds by induction on n. In the induction step one has to show that $f: \{0, \ldots, n+1\} \to \{0, \ldots, n\}$ is not injective. If there exits an $i \leq n$ such that $f(i) = f(n+1)$, then the claim is proved. Suppose $f(n+1)$ is different from all previous values. If $f(n+1) = n$, then the induction hypothesis applies to f restricted to $\{0, \ldots, n\}$. If $f(n+1) < n$, consider a function g defined on $\{0, \ldots, n\}$ where $g(i) = f(n+1)$ if $f(i) = n$ and $g(i) = f(i)$ otherwise. It is easy to show that if g is not injective, then neither is f. Then the induction hypothesis applies to g.

Formalizing proof of the pigeonhole principle is more complicated than other projects, but it is achievable for a conscientious student.

VI. Comparison with Similar Courses

Theorem provers have been used in education over the last twenty years. One of the most interesting projects is the electronic textbook Software Foundations [14] by B.C. Pierce et al. This is an introduction to formal methods used for producing reliable software. This book, which currently consists of six volumes, covers a multitude of topics: propositional and predicate logic, definitions and proofs by induction, Hoare logic, simply typed lambda calculus, functional programming, etc. A remarkable feature of the book is that all definitions, theorem and proofs are implemented in Rocq. In fact, the book is a collection of Rocq files where regular text is written in comments and the reader is invited to replay provided proofs step by step and to write their own. The main goal is not just to make readers proficient in Rocq, but to provide definitions and proofs using a new level of rigor and to demonstrate that proof assistants can be used in education along with traditional textbooks.

The paper [16] describes a system ProofWeb that underlies several online courses using Rocq. Students don't need to install Rocq on their machines; they work with the system remotely using the web interface. Each student has an account that stores passed tests, graded exercises, etc. Each course has a required supply of notes and exercises. The ProofWeb system has been used for teaching both master's level courses in verification and type theory and introductory logic courses. The authors created Rocq tactics that closely resemble the standard inference rules of natural deduction. ProofWeb can also show constructed derivations as trees, the way they are usually presented in logic courses.

An interesting Rocq-based course in the National University of Singapore is described in [17]. On the one hand, it is geared toward undergraduate students; on the other, it covers a large number of topics: propositional, predicate, modal and Hoare logics. Also, Rocq is used as a metalanguage for studying various logics as objective languages. This is an interesting approach, but it requires more than one lab a week as in our course.

The article [18] describes an attempt to teach students writing proofs beginning with fully formal ones and gradually moving towards textbook-style proofs.

A curious project is described in [19]. It combined a popular dynamic geometry software GeoGebra and Rocq. Interface with Rocq is implemented as a window inside GeoGebra. A user can draw a configuration of points and lines and form a hypothesis saying, for example, that two segments have equal lengths. The user can then experimentally confirm the hypothesis by moving free vertices and observing whether the statement remains true. Then the configuration can be transferred from GeoGebra to Rocq, and the user can write a formal proof of the hypothesis using geometric theorems from the library.

Other theorem provers are also used in education. For example, a project using the Isabelle proof assistant for teaching programming language semantics is described in [20].

VII. Conclusion and Future Work

This article describes our experience introducing mathematics majors to formal methods using the interactive proof assistant Rocq. We have shown how this gives a fresh look on proofs and helps develop a different set of skills than those cultivated by other math disciplines.

The experience shows that it is possible to teach the basics of working with a state-of-the-art proof assistant despite the course's limited length. It helps that mathematical logic is studied during the previous semester, so it is possible to leverage students' familiarity with concepts like the syntax of first-order formulas and natural deduction. Our course minimizes nontrivial aspects of Rocq, such as complex inductive types and predicates. Of course, inductive predicates such as $\leq$ on natural numbers are used extensively, but one works with them using theorems from the standard library and automatic tactics rather than their inductive definitions. Of all numerical types only natural numbers are used. Arrays are modeled by functions on natural numbers. The Curry-Howard correspondence, which is the foundation of Rocq, can be described as time permits.

Teaching this course for seven years has shown that students are usually able to prove at least a partial specification of an algorithm as a final project. Still, despite the fact that technically these proofs are quite simpler than those, say, from bachelor's theses, many students experience difficulties not just with writing proofs in Rocq, but with constructing detailed proofs on paper as well.

We plan to further develop the course by creating slides and accompanying course notes. They may allow condensing the presentation and using the freed time to describe simple inductive types, such as lists, and their corresponding induction principles. Another possible direction is studying not interactive but fully automatic systems for proving program correctness. For this purpose one can use the Why3 tool [21], which is based on Hoare logic and the calculus of weakest preconditions. Why3 can work with a large number of automatic provers (most of which work with first-order logic, unlike Rocq) and is also widely used both in academia and in the industry.

REFERENCES

[1] E. Makarov. (2025). Algorithm verification using the interactive theorem prover Rocq, [Online]. Available: https://evgenymakarov.github.io/unnfcs2019/.

[2] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, 1996. DOI: 10.1145/242223.242257.

[3] E. Feron, "Formal methods for aerospace applications," in *Formal Methods in Computer-Aided Design. FMCAD 2012*, IEEE, 2012, p. 3.

[4] W. A. Hunt Jr., M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philos Trans A Math Phys Eng Sci.*, vol. 375, no. 2104, 2017. DOI: 10.1098/rsta.2015.0399.

[5] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, 2015. DOI: 10.1145/2699417.

[6] F. Wiedijk, *The Seventeen Provers of the World*, ser. Lecture Notes in Artificial Intelligence. Berlin, Heidelberg: Springer-Verlag, 2006, vol. 3600. DOI: 10.1007/11542384.

[7] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. DOI: 10.1007/978-3-662-07964-5.

[8] The Rocq Development Team, *The Rocq Reference Manual*, version 9.0.0. INRIA, 2025. [Online]. Available: https://rocq-prover.org/doc/V9.0.0/refman/index.html.

[9] E. Makarov and B. Spitters, "The Picard algorithm for ordinary differential equations in Coq," in *Interactive Theorem Proving. ITP 2013*, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., ser. Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 463–468. DOI: 10.1007/978-3-642-39634-2_34.

[10] G. Gonthier, "Formal proof—the four-color theorem," *Notices Amer. Math. Soc.*, vol. 55, no. 11, pp. 1382–1393, 2008.

[11] R. O'Connor, "Essential incompleteness of arithmetic verified by Coq," in *Theorem Proving in Higher Order Logics. TPHOLs 2005*, J. Hurd and T. Melham, Eds., ser. Lecture Notes in Computer Science, vol. 3603, Springer, 2005, pp. 245–260. DOI: 10.1007/11541868_16.

[12] G. Gonthier et al., "Machine-checked proof of the odd order theorem," in *4th Conference on Interactive Theorem Proving, TP 2013*, ser. Lecture Notes in Computer Science, vol. 7998, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

[13] R. Krebbers, X. Leroy, and F. Wiedijk, "Formal C semantics: CompCert and the C standard," in *Interactive Theorem Proving. ITP 2014*, G. Klein and R. Gamboa, Eds., ser. Lecture Notes in Computer Science, vol. 8558, Springer, 2014, pp. 543–548. DOI: 10.1007/978-3-319-08970-6_36.

[14] B. C. Pierce et al., *Software Foundations*. Electronic textbook, 2025. [Online]. Available: https://softwarefoundations.cis.upenn.edu/.

[15] B. Russell, *Introduction to Mathematical Philosophy*. Dover Publications, 1993.

[16] M. Hendriks, C. Kaliszyk, F. van Raamsdonk, and F. Wiedijk, "Teaching logic using a state-of-the-art proof assistant," *Acta Didactica Napocensia*, vol. 3, no. 2, pp. 35–48, 2010.

[17] M. Henz and A. Hobor, "Teaching experience: Logic and formal methods with Coq," in *Certified Programs and Proofs*, J.-P. Jouannaud and Z. Shao, Eds., Springer, 2011, pp. 199–215. DOI: 10.1007/978-3-642-25379-9_16.

[18] S. Böhne and C. Kreitz, "Learning how to prove: From the Coq proof assistant to textbook style," in *Proceedings 6th International Workshop on Theorem Proving Components for Educational Software*, P. Quaresma and W. Neuper, Eds., ser. Electronic Proceedings in Theoretical Computer Science, vol. 267, Open Publishing Association, 2018, pp. 1–18. DOI: 10.4204/EPTCS.267.1.

[19] T. M. Pham and Y. Bertot, "A combination of a dynamic geometry software with a proof assistant for interactive formal proofs," *Electronic Notes in Theoretical Computer Science*, vol. 285, pp. 43–55, 2012. DOI: 10.1016/j.entcs.2012.06.005.

[20] T. Nipkow, "Teaching semantics with a proof assistant: No more LSD trip proofs," in *Verification, Model Checking, and Abstract Interpretation. VMCAI 2012*, V. Kuncak and A. Rybalchenko, Eds., ser. Lecture Notes in Computer Science, vol. 7148, 2012, pp. 24–38. DOI: 10.1007/978-3-642-27940-9_3.

[21] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Proceedings of the 22nd European Symposium on Programming*, M. Felleisen and P. Gardner, Eds., ser. Lecture Notes in Computer Science, vol. 7792, Springer, Mar. 2013, pp. 125–128. [Online]. Available: https://hal.inria.fr/hal-00789533.