Proofs-as-programs for programmers

Pierre Corbineau[1] and Jean-François Monin[1]

Verimag, University Grenoble Alpes, Grenoble INP, CNRS
pierre.corbineau@univ-grenoble-alpes.fr
jean-francois.monin@univ-grenoble-alpes.fr

Abstract

We describe our experience in teaching, with the support of the Coq proof-assistant, some skills in rigorous and clear reasoning that are useful in programming activities. This teaching was developed since 2017 within a school of computer engineers, where we do not expect students to have any particular background in mathematics. However, we do rely on intuitions from programming, particularly regarding data structures and recursion. In this context, embracing the proofs-as-programs paradigm as early as possible and without embarrassment turned out to be a challenging but interesting design decision.

1 Introduction

Mathematical skills are a must for understanding the world surrounding us and (hopefully) improving it. This is our belief as scientists and computer science is not an exception, at least in principle. But mathematics is often considered as a difficult subject. Many students, besides those who precisely wish to specialize in mathematics or areas like theoretical physics, hardly consider it as an enjoyable activity. Students in computing are much more motivated by creating software and programs than by writing pencil-and-paper proofs of abstract formulas.

How to overcome this handicap?

On the other hand a program written in their favorite programming language is nothing but a long formal expression that is accepted by the machine and returns the expected outputs only if it was developed with extreme precision down to the smallest detail. "Obvious" elliptic sentences are not included in programming languages. So programmers are used to express their creativity in a very constrained universe and they know (or should know) that if they don't understand what they are writing the chances to get a good result are low.

One of the main points in introducing computer-aided formal methods and clean functional programming languages to students in a school of computer engineers, beyond showing them that software development is amenable to a very rigorous practice especially relevant (and successful) in critical software, is to improve their ability at reasoning with a clear mind on the objects of their daily professional life such as programming constructs.

In this context, the use of a proof-assistant such as Coq has several interests: it operates as a teacher-assistant, interactively raising small challenges, like a serious game; it makes it almost impossible to cheat,[1] because all steps are checked in a strict way; it features a natural ability to describe programming languages and their semantics, as well as compilation mechanisms, which are among the fundamental topics belonging to the culture of programming engineering; if properly used, it allows to *demystify* the mathematical activity consisting in designing good definitions, expectations formulated as formal statements and developing rigorous proofs; last but not least, the very fact that the type theory supported by Coq is also a programming language makes it a tool of choice for students in computer science.

[1] More on this in Section 4.

This all looks good so far, but we would also go quickly to motivating and interesting results. Moreover, in order to favor demystification, we consider that the teacher should resist as much as possible to the temptation of cheating herself or himself with mysterious commands or notions: everything should be explained in elementary terms. This makes the implicit contract between teacher and student more balanced.

Here the use of Coq may seem more questionable, because to understand this tool, a bunch of intertwined concepts is needed and we certainly cannot afford to start with a dry exposition of the rules of the type theory implemented by Coq. We argue below that this challenge can be smoothly overcome if our teaching relies on a background and intuitions familiar to programmers, in particular data structures and recursion. This has led us to occasionally deviate, especially at the beginning, from the conventional order for presenting the main concepts of the Coq proof assistant.

In the sequel we indifferently use the worlds *function* and *program* for *functional program*.

The rest of the paper is organized as follows. Section 2 highlights a difference in approach between how to organize concepts in computer science and mathematics. Section 3 presents how we introduce Coq and its application to semantics of programming language to the students of our school of computer engineers. Section 4 concludes with observations on what could be achieved.

Credits and acknowledgments. The Coq community has already excellent resources for getting started in Coq, notably the Coq'Art [1] and Software Foundation [4],[2] which combines teaching Coq and semantics of programming languages for a long time already. The second author himself organized or participated to several training courses in Coq in an industrial or academic environment since the end of the 1990s. For various reasons it appeared to him that the existing supports were insufficiently adapted to the target audience indicated above, also taking into account the limited time available. The contents described here was then redesigned from scratch. The authors wish to thank Pascal Fradet and Maxime Lesourd who contributed to setting up this course.

2 Coq for programmers

Software or computer engineers don't even see data structures in the same way as mathematicians, at least at first sight, Their native integers are actually bounded integers. For implementing data structures (possibly: so-called "big nums"), they have in mind a linear memory model, enabling them to easily encode tree-like data structures using pointers, together with memory allocation and release. But it is clear that, when designing programs, they abstract away from those low-level considerations.

Of course, encodings of tree-like data structures are also available in traditional mathematics, using unbounded natural numbers and/or sets. Dealing with them in an effective way requires as well to abstract away from low-level details. So, whatever the encoding or implementation, the point is that tree-like structures are always conceptually obtained by repeating *simple* patterns that are easy to *visualize* (and can be implemented).

More importantly, software engineers *don't attach the same importance* to data structures as mathematicians. The latter are interested in much more abstract notions, while dealing efficiently with concrete pieces of data is an essential part of programming.

[2] See also software foundation.

We don't claim that the conceptual model used by software engineers is better than a mathematical one. We just claim that it is no less legitimate, and relevant for teaching students in computer science.

Opportunely, one of the main strengths of functional programming, besides functions as first-class values, is precisely to offer garbage collection for relieving the programmer of memory management tasks; in combination with pattern-matching, in a typed setting, we get a natural, elegant, easy to catch and powerful framework for writing (recursive) programs dealing with such data structures, and (inductively) reason about them.

3 Progression in the introduction of concepts

To summarize the previous section, we rely on the intuitive notion of tree-like data structures that our students have in mind, and we primarily present Coq as a functional programming language. A little bit of new material concerning functions is needed first, because the specific syntactic conventions coming from lambda-calculus are pervasive in our setting.

Notions are roughly introduced in the following order (slight variations are possible). It is of course a partial order, we hope that the numbering of items is self-explanatory. Note that the first items are presented here at a fairly fine level of detail. It takes between two and three weeks for items 1 to 8 and >5, where each week contains a 90 minutes lecture and a 90 minutes exercise session.

1. Basics on functional programming: anonymous functions as ordinary values, notation, functions as input and functions as output of functions
2. User-defined enumerated types: colors; basic pattern-matching.
3. Basic user-defined functions.
4. Equalities, implications, universally quantified statements; equational proofs on basic user-defined functions; tactics `intro`, `cbn`, `reflexivity`, `rewrite`, `destruct`.
5. Interactive development of functions with `intro` and the tactic `refine`.
6. Proofs as functional programs.
7. User-defined sum types (inductive types without recursion), pattern-matching.
8. User-defined inductive data types with recursion: binary trees, ASTs.

>5. Functions returning a type (introduction to dependent types and to the conversion rule).

9. Basic user-defined structural recursive functions on binary trees and ASTs

>9. Functional semantics of (ASTs of) expressions made of constants and binary operations.

10. Proofs by induction on basic user-defined structural recursive functions, tactic `induction`.
11. Proofs by induction as structural recursive programs.
12. Monomorphic lists, `append` and its basic properties; application to `reverse`.
13. Linear-time version of `reverse`.

>12. Polymorphic lists, implicit parameters.

14. Induction on a quantified formula.

a>14. Functional semantics (ASTs of) expressions containing variables.

b>14. ASTs of WHILE programs; attempt to define its functional semantics.

15. Peano natural numbers, basic properties of addition.

16. Inductive predicates and relations; proof-trees; inductive presentation of (possibly partial) functions.[3]

16+ Equality.

a>16 Natural semantics of WHILE programs.

b>16 Application of (a): correctness properties of simple programs.

17. Proofs by induction on (proofs of) inductive relations.

>17 Application: reasoning on program transformations.

18. Dealing with "absurd" hypotheses:

    (a) "Contagious" equalities.

    (b) "Super-contagious" equalities, tactic `discriminate`.

    (c) Small inversions, tactic `inversion`.

>18 Applications to semantics.

19. Structural operational semantics (SOS).

20. Certified compilation of arithmetic expressions. Extraction of correct-by-construction programs.

3.1 Comments

3.1.1 Choice of basic types to work with

Natural numbers are *not* a primitive concept of the Type Theory of Coq.[4] Moreover, programming on Peano natural numbers is not especially impressive. Booleans are a poor enumerated type, and moreover they could be misleading because the logical aspects of Coq are related to Proof Theory, not to truth values. (There is no hope to prove $\forall x, P\ x$ using Boolean algebra, as soon as x ranges over a infinite domain).

On the other hand, (finite or recursive) *inductive types* and *pattern matching* are an essential feature of typed functional programming. On the logical side they are at the root of *reasoning by cases*.

For these reasons, at the very beginning, we start with a simple enumerated type describing the three traffic light colors, the easy function returning the next color as in everyday life, and a simple theorem stating that computing three times the next color returns the original color. This is enough to illustrate items 1 to 6.

[3] In the set-theoretical sense, here.

[4] Except, maybe, when going to the hierarchy of universes; but these subtleties is far beyond the scope of these lectures.

Natural numbers and booleans are required when we come to ASTs of expressions and their evaluation (items 8 and 9), actually after one or two lectures and corresponding exercise sessions (item 8 needs only items 1 to 3 and 7). They are only used there as familiar `int`s and `bool`s (we informally explain that the formal definitions are not important at this stage and will come later for completeness). This is actually harmless since we don't need to reason on their structure at this stage.

3.1.2 Choice of the first recursive inductive types to work with

The usual approach consists in starting with simple Peano natural numbers, then progressively introducing lists and more complex inductive types. The risk is that at the end of the first days, students still get bogged down in technical details without having produced any amazing result, and then get deceived or unmotivated.

We choose to proceed in the other direction, considering that computer scientists have no conceptual problem with tree-like structures. So our first recursive inductive type is the type of binary trees (labeled with colors). After one or two introductory recursive functions on them, we consider a function for reversing a tree (recursively exchanging the left and right sub-trees). Induction is justified in the same way as structural recursion, there is no need to refer to numerical values corresponding to a size or a height. The formal proof takes then 3 straightforward lines, in contrast with the corresponding statement on lists. It is rewarding: here we can explain that writing the corresponding program in their favorite imperative programming language and proving the same intuitive property is immeasurably more complicated.

Our second recursive inductive type is for ASTs of (closed) arithmetic expressions. Their semantics is easy to define using structural recursion, and then several simple and interesting exercises can be proposed. For instance, a certified optimization performed at compile time (removing zeros on source code) is studied in our second exercise session.

Next we can come to monomorphic lists and to other considerations, such as the identification and development of auxiliary lemmas in the same way as they are used to identify and develop auxiliary functions. Lists are also a good place to introduce an important proof technique: induction on universally quantified formulas (item 14), which arise naturally when programming with accumulators.

Finally, introducing Peano natural numbers is only a matter of easy training exercises, pointing out that usual mathematical induction is just a special case of structural induction (item 15).

3.1.3 Proofs-as-programs and conversely

This paradigm is exposed very early, first on implication and universal quantification (items 6) and then for induction (item 11). We actually start in the converse way, by showing that functions can be developed interactively (items 5). From time to time, we have opportunities to directly write or to print proofs, because the corresponding functions are simple enough. In practice of course, proofs of theorems are developed interactively, but their interpretation as functions can be very useful:

1. It explains the meaning of the specialization of a quantified and/or implicate statement, as just function application.

2. Occasionally, reasoning by cases is more easily expressed using a `refine` containing an appropriate pattern-matching.

3. It is an essential part of Type Theory and error messages returned by Coq are often better understood with this paradigm in mind.

Note also that the picture is completed in 16: the proof-trees obtained there can be seen as proofs-as-inductive-data-types, and a piece of data is nothing more than a program that does not expect any arguments.

3.1.4 Dependent types

Dependent types are another key feature of the Type Theory of Coq. The point here is to make students becoming familiar with the idea that types can be (to a large extent) handled as ordinary data, before coming to polymorphism and inductively defined relations. This can be introduced very early, with a first function f taking a color as argument and returning a type such as `color` or `nat` or `color -> color`, then a second function g taking a color c as argument and returning a value in $f\,c$. Those functions are more amazing than really useful, but they illustrate that we can write well-typed functions outside the scope of usual programming languages. A little bit later we can illustrate on them the conversion rule (another key aspect of Type Theory). Similar techniques are used when we come to `discriminate` and inversion. The knowledgeable reader will recognize that the pattern-matching performed in f above is actually a *large elimination*.

3.1.5 Polymorphism

Polymorphism is introduced quite late. The reason is that most new and challenging definitions are easier to state with monomorphic data structures. The generalization to their polymorphic version is conceptually straightforward but adds complications if introduced too early: explicit terms become clumsy to write and read, so that in practice implicit arguments need to be introduced and explained as well. In other words, the risk is to heal the unfortunate effects of a distracting notion (as far as we can avoid it for a while) with yet another distracting notion. In summary we prefer to postpone the introduction of these technicalities to a stage where it can be done quickly and without pain, when students are familiar with the fact that types can be manipulated like ordinary values.

3.1.6 Equality

The first properties of functions considered in the introductory part are naturally expressed with equalities. At this stage, as in *Software Foundations*, we just use this relation, which behaves just as expected whatever its formal definition. For completeness, the definition of equality can be presented once inductive relations (and polymorphic definitions) are known. Note that, in the more advanced part of the course dedicated to big-steps and small-steps operational semantics, the main properties are not stated with equalities but with user-defined inductive relations.

3.1.7 Inversion

We strive to explain everything in basic terms, including very useful tactics such as `discriminate`, `injection` and `inversion` that are better demystified. The former is implemented using large eliminations. Actually a weaker elimination can be used in many situations, if we remark that equalizing two values obtained by different constructors C_1 and C_2 (applied to suitable arguments if needed) allows us to infer that in any type T, all values are equal. To

see this, given $x, y : T$, consider a pattern-matching returning respectively x and y for C_1 and C_2 (applied to their above arguments). The trick still works for proving arbitrary predicates applied to any arguments as soon as they can be proved for some arguments. When such witnesses are not available, or for arbitrary propositions, the same trick can be used but with a large elimination, as performed systematically by the standard tactic `discriminate`.

For inversions, we detail a variant of *small inversions* [2] briefly explained in [3]. Note that here again we have a conceptually simple but essential use of large eliminations.

Here we only expect the best students to follow the details. But all should be able to recognize situations where an inversion is relevant, and understand that the standard tactic can, in principle, be explained without "black magic".

3.1.8 Logic

We don't insist on the subtle differences (or relations) between constructive and classical logic: we have no opportunity to talk about excluded middle or double negation elimination since negation itself is never used in the targeted applications, all the theorems considered happen to be stated in a positive way. (Even "absurd" hypotheses can be dealt with in a positive way to a large extent, see 3.1.7; this is why we prefer the term "contagious equality").

More surprisingly, we even discovered, just by trying to systematically remove unused notions in order to save time, that we can even dispense with conjunction and disjunction. A good news, since their definition is polymorphic. This comes from the fact that propositions are naturally combined using n-ary choices between sequences of quantifications and implications (telescopes), often with recursion, so that user-defined inductive relations are naturally appropriate. The basic propositions `True` and `False` are not really needed – which is not that bad, to avoid confusion with Boolean values.

Of course, we also keep usual basic material on propositional and first order logic that can be presented, depending on time and occasional needs. But it is not necessary (for this course) to start with it.

4 Assessment

The contents presented here is presented in approximately 30-35 hours. The support material (in French) is available at `https://www-verimag.imag.fr/~monin/Enseignement/LTPF/robuste/LT/index.html`.[5] We are rather strict in checking that students do their homework regularly, and most importantly that they *understand* what they write: at the end, no final hard copy exam, but a project gathering the most important topics followed by an oral individual examination where they are expected to explain their own answers. This project is shared with another course on functional programming (in OCaml), with a similar volume. Though the latter is dedicated to other aspects, such as programming techniques or parsing, there is clearly a synergy between these two courses. According to the anonymous feedback provided by the students in the last years and to the results obtained, ranging evenly from "not lost" to "was able to solve challenging problems" for the most excited, this experience in teaching Coq looks reasonably successful.

Of course one has to be very careful at making sure that all the material needed for a given exercise was presented and understood earlier. Questions to *stackoverflow* are likely to provide

[5] In April 2024, the second author gave a Coq course at ECNU, Shanghai, based on an English translation completed by some more advanced material. See `https://www-verimag.imag.fr/~monin/Enseignement/ECNU-2024/`.

poor results for beginners.

Up to this precondition, the main conclusion we draw from this experiment is that there is more freedom than one may expect at first sight for teaching Coq and applications. In particular relying on the background of programmers can be interesting.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[2] Jean-François Monin and Xiaomu Shi. Handcrafted Inversions Made Operational on Operational Semantics. In S. Blazy, C. Paulin, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 338–353, Rennes, France, July 2013. Springer.

[3] Jean-François Monin. Small inversions for smaller inversions. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022 Abstracts)*, Nantes, June 2022.

[4] Benjamin C. Pierce. Software foundations, 15 years on, July 2022. Talk at Newton Institute workshop on Formal Education.