1

A Brief Overview

Coq [36] is a proof assistant with which students, researchers, or engineers can express specifications and develop programs that fulfill these specifications. This tool is well adapted to develop programs for which absolute trust is required: for example, in telecommunication, transportation, energy, banking, etc. In these domains the need for programs that rigorously conform to specifications justifies the effort required to verify these programs formally. We shall see in this book how a *proof assistant* like *Coq* can make this work easier.

The *Coq* system is not only interesting to develop safe programs. It is also a system with which mathematicians can develop proofs in a very expressive logic, often called *higher-order logic*. These proofs are built in an interactive manner with the aid of automatic search tools when possible. The application domains are very numerous, for instance logic, automata theory, computational linguistics and algorithmics (see [83]).

This system can also be used as a logical framework to give the axioms of new logics and to develop proofs in these logics. For instance, it can be used to implement reasoning systems for modal logics, temporal logics, resource-oriented logics, or reasoning systems on imperative programs.

The *Coq* system belongs to a large family of computer-based tools whose purpose is to help in proving theorems, namely *Automath* [33], *Nqthm* [16, 17], *Mizar* [82], *LCF* [47], *Nuprl* [24], *Isabelle* [72], *Lego* [59], *HOL* [46], *PVS* [67], and *ACL2* [54], which are other renowned members of this family. A remarkable characteristic of *Coq* is the possibility to generate certified programs from the proofs and, more recently, certified modules.

In this introductory chapter, we want to present informally the main features of *Coq*. Rigorous definitions and precise notation are given in later chapters and we only use notation taken from usual mathematical practice or from programming languages.

1.1 Expressions, Types, and Functions

The specification language of *Coq*, also called *Gallina*, makes it possible to represent the usual types and programs of programming languages.

Expressions in this language are formed with constants and identifiers, following a few construction rules. Every expression has a type; the type for an identifier is usually given by a *declaration* and the rules that make it possible to form combined expressions come with *typing rules* that express the links between the type of the parts and the type of the whole expression.

For instance, let us consider the type `Z` of integers, corresponding to the set $\mathbb{Z}$. The constant -6 has this type and if one declares a variable z with type `Z`, the expression $-6z$ also has type `Z`. On the other hand, the constant `true` has type `bool` and the expression "`true` $\times -6$" is not a well-formed expression.

We can find a large variety of types in the *Gallina* language: besides `Z` and `bool`, we make intensive use of the type `nat` of natural numbers, considered as the smallest type containing the constant 0 and the values obtained when calling the successor function. Type operators also make it possible to construct the type $A \times B$ of pairs of values (a, b) where a has type A and b has type B, the type "`list` A" of lists where all elements have type A and the type $A \to B$ of functions mapping any argument of type A to a result of type B.

For instance, the functional that maps any function f from `nat` to `Z` and any natural number n to the value $\Sigma_{i=0}^{i=n} f(i)$ can be defined in *Gallina* and has type `(nat→Z)→nat→Z`.

We must emphasize that we consider the notion of *function* from a computer science point of view: functions are effective computing processes (in other words, *algorithms*) mapping values of type A to values of type B; this point of view differs from the point of view of set theory, where functions are particular subsets of the cartesian product $A \times B$.

In *Coq*, computing a value is done by successive reductions of terms to an irreducible form. A fundamental property of the *Coq* formalism is that computation always terminates (a property known as *strong normalization*). Classical results on computability show that a programming language that makes it possible to describe all computable functions must contain functions whose computations do not terminate. For this reason, there are computable functions that can be described in *Coq* but for which the computation cannot be performed by the reduction mechanism. In spite of this limitation, the typing system of *Coq* is powerful enough to make it possible to describe a large subclass of all computable functions. Imposing strong normalization does not significantly reduce the expressive power.

1.2 Propositions and Proofs

The *Coq* system is not just another programming language. It actually makes it possible to express assertions about the values being manipulated. These values may range over mathematical objects or over programs.

Here are a few examples of assertions or *propositions*:

- $3 \leq 8$,
- $8 \leq 3$,
- "for all $n \geq 2$ the sequence of integers defined by

$$u_0 = n$$
$$u_{i+1} = \begin{cases} 1 & \text{when } u_i = 1 \\ u_i/2 & \text{when } u_i \text{ is even} \\ 3u_i + 1 & \text{otherwise} \end{cases}$$

  ultimately reaches the value 1,"
- "list concatenation is associative,"
- "the algorithm `insertion_sort` is a correct sorting method."

Some of these assertions are true, others are false, and some are still conjectures—the third assertion[1] is one such example. Nevertheless, all these examples are well-formed propositions in the proper context.

To make sure a proposition P is true, a safe approach is to provide a proof. If this proof is complete and readable it can be verified. These requirements are seldom satisfied, even in the scientific literature. Inherent ambiguities in all natural languages make it difficult to verify that a proof is correct. Also, the complete proof of a theorem quickly becomes a huge text and many reasoning steps are often removed to make the text more readable.

A possible solution to this problem is to define a formal language for proofs, built along precise rules taken from proof theory. This makes it possible to ensure that every proof can be verified step by step.

The size of complete proofs makes it necessary to mechanize their verification. To trust such a mechanical verification process, it is enough to show that the verification algorithm actually verifies that all formal rules are correctly applied.

The size of complete proofs also makes it inpractical to write them manually. In this sense, naming a tool like *Coq* a *proof assistant* becomes very meaningful. Given a proposition that one wants to prove, the system proposes tools to construct a proof. These tools, called *tactics*, make it easier to construct the proof of a proposition, using elements taken from a context, namely, declarations, definitions, axioms, hypotheses, lemmas, and theorems that were already proven.

In many cases, tactics make it possible to construct proofs automatically, but this cannot always be the case. Classical results on proof complexity and

[1] The sequence u_i in this assertion is known as the "Syracuse sequence."

computability show that it is impossible to design a general algorithm that can build a proof for every true formula. For this reason, the *Coq* system is an interactive system where the user is given the possibility to decompose a difficult proof in a collection of lemmas, and to choose the tactic that is adapted to a difficult case. There is a wide variety of available tactics and expert users also have the possibility to add their own tactics (see Sect. 7.6).

The user can actually choose not to read proofs, relying on the existence of automatic tools to construct them and a safe mechanism to verify them.

1.3 Propositions and Types

What is a good language to write proofs? Following a tradition that dates back to the *Automath* project [33], one can write the proofs and programs in the same formalism: *typed λ-calculus*. This formalism, invented by Church [23], is one of the many formalisms that can be used to describe computable algorithms and directly inspired the design of all programming languages in the *ML* family. The *Coq* system uses a very expressive variation on typed λ-calculus, the *Calculus of Inductive Constructions* [27, 69].[2]

Chapter 3 of this book covers the relation between proofs and programs, generally called the *Curry–Howard isomorphism*. The relation between a program and its type is the same as the relation between a proof and the statement it proves. Thus verifying a proof is done by a type verification algorithm. Throughout this book, we shall see that the practice of *Coq* is made easier thanks to the double knowledge mixing intuitions from functional programming and from reasoning practice.

An important characteristic of the Calculus of Constructions is that every type is also a term and also has a type. The type of a proposition is called `Prop`. For instance, the proposition $3 \leq 7$ is at the same time the type of all proofs that 3 is smaller than 7 and a term of type `Prop`.

In the same spirit, a *predicate* makes it possible to build a parametric proposition. For instance, the predicate "to be a prime number" enables us to form propositions: "7 is a prime number," "1024 is a prime number," and so on. This predicate can then be considered as a function, whose type is `nat→Prop` (see Chap. 4). Other examples of predicates are the predicate "to be a sorted list" with type `(list Z)→Prop` and the binary relation $\leq$, with type `Z→Z→Prop`.

More complex predicates can be described in *Coq* because arguments may themselves be predicates. For instance, the property of being a transitive relation on `Z` is a predicate of type `(Z→Z→Prop)→Prop`. It is even possible to consider a *polymorphic* notion of transitivity with the following type:

`(`A`→`A`→Prop)→Prop` *for every data type* A.

[2] We sometimes say *Calculus of Constructions* for short.

1.4 Specifications and Certified Programs

The few examples we have already seen show that propositions can refer to data and programs.

The *Coq* type system also makes it possible to consider the converse: the type of a program can contain constraints expressed as propositions that must be satisfied by the data. For instance, if n has type `nat`, the type "a prime number that divides n" contains a *computation-related* part "a value of type `nat`" and a *logical* part "this value is prime and divides n."

This kind of type, called a *dependent* type because it depends on n, contributes to a large extent to the expressive power of the *Coq* language. Other examples of dependent types are data structures containing size constraints: vectors of fixed length, trees of fixed height, and so on.

In the same spirit, the type of functions that map any $n > 1$ to a prime divisor of n can be described in *Coq* (see Chap. 9). Functions of this type all compute a prime divisor of the input as soon as the input satisfies the constraint. These functions can be built with the interactive help of the *Coq* system. It is called a certified program and contains both computing information and a proof: that is, how to compute such a prime divisor and the reason why the resulting number actually is a prime number dividing n.

An *extraction* algorithm makes it possible to obtain an *OCAML* [22] program that can be compiled and executed from a certified program. Such a program, obtained mechanically from the proof that a specification can be fulfilled, provides an optimal level of safety. The extraction algorithm works by removing all logical arguments to keep only the description of the computation to perform. This makes the distinction between computational and logical information important. This extraction algorithm is presented in Chaps. 10 and 11.

1.5 A Sorting Example

In this section, we informally present an example to illustrate the use of *Coq* in the development of a certified program. The reader can download the complete *Coq* source from the site of this book [9], but it is also given in the appendix at the end of this book. We consider the type "`list Z`" of the lists of elements of type `Z`. We (temporarily) use the following notation: the empty list is written `nil`, the list containing 1, 5, and -36 is written `1::5::-36::nil`, and the result of adding n in front of the list l is written $n :: l$.

How can we specify a sorting program? Such a program is a function that maps any list l of the type "`list Z`" to a list l' where all elements are placed in increasing order and where all the elements of l are present, also respecting the number of times they occur. Such properties can be formalized with two predicates whose definitions are described below.

1.5.1 Inductive Definitions

To define the predicate "to be a sorted list," we can use the *Prolog* language as inspiration. In *Prolog*, we can define a predicate with the help of clauses that enumerate sufficient conditions for this predicate to be satisfied. In our case, we consider three clauses:

1. the empty list is sorted,
2. every list with only one element is sorted,
3. if a list of the form $n :: l$ is sorted and if $p \leq n$ then the list $p :: n :: l$ is sorted.

In other words, we consider the smallest subset X of "`list Z`" that contains the empty list, all lists with only one element, and such that if the list $n :: l$ is in X and $p \leq n$ then $p :: n :: l \in X$. This kind of definition is given as an *inductive definition* for a predicate `sorted` using three constructing rules (corresponding to the clauses in a *Prolog* program).

Inductive sorted:`list Z`→`Prop`:=
sorted0: sorted(nil)
sorted1: $\forall z : \texttt{Z},\ \text{sorted}(z :: nil)$
sorted2: $\forall z_1, z_2 : \texttt{Z},\ \forall l : \texttt{list Z},\ z_1 \leq z_2 \Rightarrow \text{sorted}(z_2 :: l) \Rightarrow \text{sorted}(z_1 :: z_2 :: l)$

This kind of definition is studied in Chaps. 8 and 14.

Proving, for instance, that the list `3::6::9::nil` is sorted is easy thanks to the construction rules. Reasoning about arbitrary sorted lists is also possible thanks to associated lemmas that are automatically generated by the *Coq* system. For instance, techniques known as *inversion techniques* (see Sect. 8.5.2) make it possible to prove the following lemma:

$$\textbf{sorted\_inv: } \forall z : \texttt{Z},\ \forall l : \texttt{list Z},\ \text{sorted}(z :: l) \Rightarrow \text{sorted}(l)$$

1.5.2 The Relation "to have the same elements"

It remains to define a binary relation expressing that a list l is a permutation of another list l'. A simple way is to define a function `nb_occ` of type "`Z`→`list Z`→`nat`" which maps any number z and list l to the number of times that z occurs in l. This function can simply be defined as a recursive function. In a second step we can define the following binary relation on lists of elements in `Z`:

$$l \equiv l' \Leftrightarrow \forall z : \texttt{Z}, \texttt{nb\_occ}\ z\ l = \texttt{nb\_occ}\ z\ l'$$

This definition does not provide a way to determine whether two lists are permutations of each other. Actually, trying to follow it naïvely would require comparing the number of occurrences of z in l and l' for all members of $\mathbb{Z}$ and this set is infinite! Nevertheless, it is easy to prove that the relation $\equiv$ is an equivalence relation and that it satisfies the following properties:

equiv_cons: $\forall z : \texttt{Z},\ \forall l, l' : \texttt{list Z},\ l \equiv l' \Rightarrow z :: l \equiv z :: l'$
equiv_perm: $\forall n, p : \texttt{Z},\ \forall l, l' : \texttt{list Z},\ l \equiv l' \Rightarrow n :: p :: l \equiv p :: n :: l'$

These lemmas will be used in the certification of a sorting program.

1.5.3 A Specification for a Sorting Program

All the elements are now available to specify a sorting function on lists of integers. We have already seen that the *Coq* type system integrates complex specifications, with which we can constrain the input and output data of programs. The specification of a sorting algorithm is the type `Z_sort` of the functions that map any list l : `list Z` to a list l' satisfying the proposition $\text{sorted}(l') \wedge l \equiv l'$.

Building a certified sorting program is the same as building a term of type `Z_sort`. In the next sections, we show how to build such a term.

1.5.4 An Auxiliary Function

For the sake of simplicity, we consider insertion sort. This algorithm relies on an auxiliary function to insert an element in an already sorted list. This function, named `aux`, has type "`Z`→`list Z`→`list Z`." We define `aux` n l in the following manner, in a recursion where l varies:

- **if** l is empty, **then** $n :: nil$,
- **if** l has the form $p :: l'$ **then**
  - **if** $n \leq p$ **then** $n :: p :: l'$,
  - **if** $p < n$, **then** $p ::$ (`aux` n l').

This definition uses a comparison between n and p. It is necessary to understand that the possibility to compare two numbers is a property of `Z`: the order $\leq$ is decidable. In other words, it is possible to program a function with two arguments n and p that returns a certain value when $n \leq p$ and a different value when $n > p$. In the *Coq* system, this property is represented by a certified program given in the standard library and called `Z_le_gt_dec` (see Sect. 9.1.3). Not every order is decidable. For instance, we can consider the type `nat`→`nat` representing the functions from $\mathbb{N}$ to $\mathbb{N}$ and the following relation:

$$f < g \Leftrightarrow \exists i \in \mathbb{N},\ f(i) < g(i) \wedge (\forall j \in \mathbb{N}.\ j < i \Rightarrow f(j) = g(j))$$

This order relation is undecidable and it is impossible to design a comparison program similar to `Z_le_gt_dec` for this order. A consequence of this is that we cannot design a program to sort lists of functions.[3] The purpose of function `aux` is described in the following two lemmas, which are easily proved by induction on l:

[3] This kind of problem is not inherent to *Coq*. When a programming language provides comparison primitives for a type A it is only because comparison is decidable in this type. The *Coq* system only underlines this situation.

aux_equiv: $\forall l : \texttt{list Z}$, $\forall n : \texttt{Z}$, $\texttt{aux n l} \equiv n :: l$,
aux_sorted: $\forall l : \texttt{list Z}$, $\forall n : \texttt{Z}$, $\texttt{sorted}\ l \Rightarrow \texttt{sorted aux}\ n\ l$

1.5.5 The Main Sorting Function

It remains to build a certified sorting program. The goal is to map any list l to a list l' that satisfies $\texttt{sorted}\ l' \wedge l \equiv l'$.

This program is defined using induction on the list l:

- If l is empty, then $l' = \texttt{[]}$ is the right value.
- Otherwise l has the form $l = n :: l_1$.
  - The induction hypothesis on l_1 expresses that we can take a list l'_1 satisfying "$\texttt{sorted}\ l'_1 \wedge l_1 \equiv l'_1$".
    Now let l' be the list $\texttt{aux}\ n\ l'_1$
  - thanks to the lemma `aux_sorted` we know $\texttt{sorted}\ l'$,
  - thanks to the lemma `aux_equiv` and `equiv_cons` we know

$$l = n :: l_1 \equiv n :: l'_1 \equiv \texttt{aux}\ n\ l'_1 = l'.$$

This construction of l' from l, with its logical justifications, is developed in a dialogue with the *Coq* sytem. The outcome is a term of type `Z_sort`, in other words, a certified sorting program. Using the extraction algorithm on this program, we obtain a functional program to sort lists of integers. Here is the output of the `Extraction` command:[4]

```
let rec aux z0 = function
  | Nil -> Cons (z0, Nil)
  | Cons (a, l') ->
      (match z_le_gt_dec z0 a with
         | Left -> Cons (z0, (Cons (a, l')))
         | Right -> Cons (a, (aux z0 l')))

let rec sort = function
  | Nil -> Nil
  | Cons (a, tl) -> aux a (sort tl)
```

This capability to construct mechanically a program from the proof that a specification can be satisfied is extremely important. Proofs of programs that could be done on a blackboard or on paper would be incomplete (because they are too long to write) and even if they were correct, the manual transcription into a program would still be an occasion to insert errors.

[4] This program uses a type with two constructors, `Left` and `Right`, that is isomorphic to the type of boolean values.

1.6 Learning *Coq*

The *Coq* system is a computer tool. To communicate with this tool, it is mandatory to obey the rules of a precise language containing a number of commands and syntactic conventions. The language that is used to describe terms, types, proofs, and programs is called *Gallina* and the command language is called *Vernacular*. The precise definition of these languages is given in the *Coq* reference manual [80].

Since *Coq* is an interactive tool, working with it is a dialogue that we have tried to transcribe in this book. The majority of the examples we give in this book are well-formed examples of using *Coq*. For pedagogical reasons, some examples also exhibit erroneous or clumsy uses and guidelines to avoid problems are also described.

The *Coq* development team also maintains a site that gathers all the contributions of users[5] with many formal developments concerning a large variety of application domains. We advise the reader to consult this repository regularly. We also advise suscribing to the `coq-club` mailing list,[6] where general questions about the evolution of the system appear, its logical formalism are discussed, and new user contributions are announced.

As well as for training on the *Coq* tool, this book is also a practical introduction to the theoretical framework of type theory and, more particularly, the Calculus of Inductive Constructions that combines several of the recent advances in logic from the point of view of λ-calculus and typing. This research field has its roots in the work of Russell and Whitehead, Peano, Church, Curry, Prawitz, and Aczel; the curious reader is invited to consult the collection of papers edited by J. van Heijenoort "From Frege to Gödel" [84].

1.7 Contents of This Book

The Calculus of Constructions

Chapters 2 to 4 describe the Calculus of Constructions. Chapter 2 presents the simply typed λ-calculus and its relation with functional programming. Important notions of terms, types, sorts, and reductions are presented in this chapter, together with the syntax used in *Coq*.

Chapter 3 introduces the logical aspects of *Coq*, mainly with the Curry–Howard isomorphism; this introduction uses the restricted framework that combines simply typed λ-calculus and minimal propositional logic. This makes it possible to introduce the notion of *tactics*, the tools that support interactive proof development.

The full expressive power of the Calculus of Constructions, encompassing polymorphism, dependent types, higher-order types, and so on, is studied

[5] `http://coq.inria.fr/contribs-eng.html`
[6] `coq-club@pauillac.inria.fr`

in Chap. 4 where the notion of *dependent product* is introduced. With this type construct, we can extend the Curry–Howard isomorphism to notions like universal quantification.

In Chap. 5, we show how the Calculus of Constructions can be used practically to model logical reasoning. This chapter shows how to perform simple proofs in a powerful logic.

Inductive Constructions

The Calculus of Inductive Constructions is introduced in Chap. 6, where we describe how to define data structures such as natural numbers, lists, and trees. New tools are associated with these types: tactics for proofs by induction, simplification rules, and so on. Chapter 7 is not related to inductive constructions but is included there because the tactics it describes make the exposition in chapter 8 easier to present.

The notion of inductive type is not restricted to tree-like data structures: predicates can also be defined inductively, in the style of some kind of higher-order *Prolog*[7] (Chap. 8). This framework also accommodates the basic notions of everyday logic: contradiction, conjunction, disjunction, existential quantification.

Certified Programs and Extraction

In Chap. 9, we study how to describe complex specifications about data and programs containing logical components. A variety of type constructs that make it possible to express a variety of program specifications is presented, together with the tactics that make the task of building certified programs easier.

Chapters 10 and 11 study how to produce some *OCAML* code effectively by mechanical extraction from the proofs.

Advanced Use

The last chapters of the book present the advanced aspects of *Coq*, both from the user's point of view and from the understanding of its theoretical mechanics.

Chapter 12 presents the module system of *Coq*, which is closely related to the module system of *OCAML*. This system makes it possible to represent the dependencies between mathematical theories and to build truly reusable certified program components.

[7] Recall that *Prolog* concerns first-order logic. In the framework of plain *Prolog*, it is not possible to define predicates over any relation, like the transitive closure of a binary relation. In the *Coq* framework, this is possible.

Chapter 13 shows how to represent infinite objects—for instance, the behaviors of transition systems—in an extension of the Calculus of Constructions. We describe the main techniques to specify and build these objects, together with proof techniques.

Chapter 14 revisits the basic principles that govern inductive definitions and ensure their logical consistency. This study provides keys for even more advanced studies.

Chapter 15 describes techniques to broaden the class of recursive functions that can be described in the Calculus of Inductive Constructions and how to reason on these functions.

Proof Automation

Chapter 7 introduces the tools that make it possible to build complex proofs: namely, tactics. This chapter describes the tactics associated with inductive types, automatic proof tactics, numerical proof tactics, and a language that makes it possible to define new tactics. Thanks to the tools provided in this chapter, the proofs described in later chapters can be described more concisely.

Chapter 16 describes a design technique for complex tactics that is particularly well-suited for the Calculus of Constructions: namely, the technique of proof by reflection.

1.8 Lexical Conventions

This book provides a wealth of examples that can be input in the *Coq* system and the answers of the system to these examples. Throughout this book, we use the classical convention of books about computer tools: the `typewriter` font is used to represent text input by the user, while the *italic* font is used to represent the text output by the system as answers. We have respected the syntax imposed by *Coq*, but we rely on a few conventions that make the text easier to read:

- The mathematical symbols $\leq$, $\neq$, $\exists$, $\forall$, $\rightarrow$, $\vee$, $\wedge$, and $\Rightarrow$ stand for the character strings `<=`, `<>`, `exists`, `forall`, `->`, `\/`, `/\`, and `=>`, respectively. For instance, the *Coq* statement

  ```
  forall A:Set,(exists x : A, forall (y:A), x <> y) -> 2 = 3
  ```

  is written as follows in this book:

  ```
  ∀A:Set,(∃x:A, ∀y:A, x ≠ y)→ 2 = 3
  ```

- When a fragment of *Coq* input text appears in the middle of regular text, we often place this fragment between double quotes "...." These double quotes do not belong to the *Coq* syntax.
- Users should also remember that any string enclosed between `(*` and `*)` is a comment and is ignored by the *Coq* system.