# Foreword

When Don Knuth undertook his masterpiece to lay the foundations of computer science in a treatise on programming, he did not choose to entitle his work “The Science of Computer Programming” but “The Art of Computer Programming.” Accordingly, it took 30 more years of research to really establish a rigorous field on programming and algorithms. In a similar fashion, the rigorous foundations of the field of formal proof design are still being laid down. Although the main concepts of proof theory date back to the work of Gentzen, Gödel, and Herbrand in the 1930s, and although Turing himself had a pioneering interest in automating the construction of mathematical proofs, it is only during the 1960s that the first experiments in automatic first-order logic by systematically enumerating the Herbrand domain took place. Forty years later, the *Coq* proof assistant is the latest product in a long series of investigations on computational logic and, in a way, it represents the state of the art in this field. However, its actual use remains a form of art, difficult to master and to improve. The book of Yves Bertot and Pierre Castéran is an invaluable guide, providing beginners with an initial training and regular practitioners with the necessary expertise for developing the mathematical proofs that are needed for real-size applications.

A short historical presentation of the *Coq* system may help to understand this software and the mathematical notions it implements. The origins of the underlying concepts may also provide clues to understanding the mechanics that the user must control, the various points of view to adopt when building a system’s model, the options to consider in case of trouble.

Gérard Huet started working on automatic theorem proving in 1970, using LISP to implement the SAM prover for first-order logic with equality. At the time, the state of the art was to translate all logical propositions into lists (conjunctions) of lists (disjunctions) of literals (signed atomic formulas), quantification being replaced by Skolem functions. In this representation deduction was reduced to a principle of pairing complementary atomic formulas modulo instantiation (so-called resolution with principal unifiers). Equalities gave rise to unidirectional rewritings, again modulo unification. Rewriting order was determined in an ad hoc way and there was no insurance that the process would converge, or whether it was complete. Provers were black boxes that generated scores of unreadable logical consequences. The standard working technique was to enter your conjecture and wait until the computer's memory was full. Only in exceptionally trivial cases was there an answer worth anything. This catastrophic situation was not recognized as such, it was understood as a necessary evil, blamed on the incompleteness theorems. Nevertheless, complexity studies would soon show that even in decidable areas, such as propositional logic, automatic theorem proving was doomed to run into a combinatorial wall.

A decisive breakthrough came in the 1970s with the implementation of a systematic methodology to use termination orders to guide rewriting, starting from the founding paper of Knuth and Bendix. The KB software, implemented in 1980 by Jean-Marie Hullot and Gérard Huet, could be used to automate in a natural way decision and semi-decision procedures for algebraic structures. At the same time, the domain of proofs by induction was also making steady progress, most notably with the NQTHM/ACL of Boyer and Moore. Another significant step had been the generalization of the resolution technique to higher-order logic, using a unification algorithm for the theory of simple types, designed by Gérard Huet back in 1972. This algorithm was consistent with a general approach to unification in an equational theory, worked out independently by Gordon Plotkin.

At the same time, logicians (Dana Scott) and theoretical computer scientists (Gordon Plotkin, Gilles Kahn, Gérard Berry) were charting a logical theory of computable functions (computational domains) together with an effectively usable axiomatization (computational induction) to define the semantics of programming languages. There was hope of using this theory to address rigorously the problem of designing trustworthy software using formal methods. The validity of a program with respect to its logical specifications could be expressed as a theorem in a mathematical theory that described the data and control structures used by the algorithm. These ideas were set to work most notably by Robin Milner's team at Edinburgh University, who implemented the LCF system around 1980. The salient feature of this system was its use of proof *tactics* that could be programmed in a meta-language (ML). The formulas were not reduced to undecipherable clauses and users could use their intuition and knowledge of the subject matter to guide the system within proofs that mixed automatic steps (combining predefined and specific tactics that users could program in the ML language) and easily understandable manual steps.

Another line of investigation was explored by the philosopher Per Martin-Löf, starting from the constructive foundations of mathematics initially proposed by Brouwer and extended notably by Bishop's development of constructive analysis. Martin-Löf's Intuitionistic Theory of Types, designed at the beginning of the 1980s, provided an elegant and general framework for the constructive axiomatization of mathematical structures, well suited to serve as a foundation for functional programming. This direction was seriously pursued by Bob Constable at Cornell University who undertook the implementation of the NuPRL software for the design of software from formal proofs, as well as by the "Programming methodology" team headed by Bengt Nordström at Chalmers University in Gothenburg.

All this research relied on the $\lambda$-calculus notation, initially designed by the logician Alonzo Church, in its pure version as a language to define recursive functionals, and in its typed version as a higher-order predicate calculus (the theory of simple types, a simpler alternative for meta-mathematics to the system originally used by Whitehead and Russell in *Principia Mathematica*). Furthermore, the $\lambda$-calculus could also be used to represent proofs in a natural deduction format, thus yielding the famous "Curry–Howard correspondence," which expresses an isomorphism between proof structures and functional spaces. These two aspects of the $\lambda$-calculus were actually used in the Automath system for the representation of mathematics, designed by Niklaus de Bruijn in Eindhoven during the 1970s. In this system, the types of $\lambda$-expressions were no longer simple hierarchical layers of functional spaces. Instead they were actually $\lambda$-expressions that could express the dependence of a functional term's result type on the value of its argument—in analogy with the extension of propositional calculus to first-order predicate calculus, where predicates take as arguments terms that represent elements of the carrier domain.

$\lambda$-calculus was indeed the main tool in proof theory. In 1970, Jean-Yves Girard proved the consistency of Analysis through a proof of termination for a polymorphic $\lambda$-calculus called system $F$. This system could be generalized to a calculus $F\omega$ with polymorphic functionals, thus making it possible to encode a class of algorithms that transcended the traditional ordinal hierarchies. The same system was to be rediscovered in 1974 by John Reynolds, as a proposal for a generic programming language that would generalize the restricted form of polymorphism that was present in ML.

In the early 1980s, research was in full swing at the frontier between logic and computer science, in a field that came to be known as Type Theory. In 1982 Gérard Huet started the Formel project at INRIA's Rocquencourt laboratory, jointly with Guy Cousineau and Pierre-Louis Curien from the computer science laboratory at École Normale Supérieure. This team set the objective of designing and developing a proof system extending the ideas of the LCF system, in particular by adopting the ML language not only as the meta-language used to define tactics but also as the implementation language of the whole proof system. This research and development effort on functional programming would lead over the years to the Caml language family and, ultimately, to its latest offspring Objective Caml, still used to this day as the implementation language for the *Coq* proof assistant.

At the international conference on types organized by Gilles Kahn in Sophia Antipolis in 1984, Thierry Coquand and Gérard Huet presented a synthesis of dependent types and polymorphism that made it possible to adapt

Martin-Löf's constructive theory to an extension of the Automath system called the *Calculus of Constructions*. In his doctoral thesis, Thierry Coquand provided a meta-theoretical analysis of the underlying $\lambda$-calculus. By proving the termination of this calculus, he also provided a proof of its logical soundness. This calculus was adopted as the logical basis for the Formel project's proof system and Gérard Huet proposed a first verifier for this calculus (CoC) using as a virtual machine his *Constructive Engine*. This verifier made it possible to present a few formal mathematical developments at the Eurocal congress in April 1985.

This was the first stage of what was to become the *Coq* system: a type verifier for $\lambda$-expressions that represent either proof terms in a logical system or the definition of mathematical objects. This proof assistant kernel was completely independent from the proof synthesis tool that was used to construct the terms to be verified—the interpreter for the constructive engine is a deterministic program. Thierry Coquand implemented a sequent-style proof synthesis algorithm that made it possible to build proof terms by progressive refinement, using a set of tactics that were inspired from the LCF system. The second stage would soon be completed by Christine Mohring, with the initial implementation of a proof-search algorithm in the style of *Prolog*, the famous `Auto` tactic. This was practically the birth of the *Coq* system as we know it today. In the current version, the kernel still rechecks the proof term that is synthesized by the tactics that are called by the user. This architecture has the extra advantage of making it possible to simplify the proof-search machinery, which actually ignores some of the constraints imposed by stratification in the type system.

The Formel team soon considered that the Calculus of Constructions could be used to synthesize certified programs, in the spirit of the NuPRL system. A key point was to take advantage of polymorphism, whose power may be used to express as a type of system $F$ an algebraic structure, such as the integers, making systematic use of a method proposed by Böhm and Berarducci. Christine Mohring concentrated on this issue and implemented a complex tactic to synthesize induction principles in the Calculus of Constructions. This allowed her to present a method for the formal development of certified algorithms at the conference "Logic in Computer Science (LICS)" in June 1986. However, when completing this study in her doctoral thesis, she realized that the "impredicative" encodings she was using did not respect the tradition where the terms of an inductive type are restricted to compositions of the type constructors. Encodings in the polymorphic $\lambda$-calculus introduced parasitic terms and made it impossible to express the appropriate inductive principles. This partial failure actually gave Christine Mohring and Thierry Coquand the motivation to design in 1988 the "Calculus of Inductive Constructions," an extension of the formalism, endowed with good properties for the axiomatization of algorithms on inductive data structures.

The Formel team was always careful to balance theoretical research and experimentation with models to assert the feasibility of the proposed ideas, prototypes to verify the scalability to real-size proofs, and more complete systems, distributed as free software, with a well-maintained library, documentation, and a conscious effort to ensure the compatibility between successive versions. The team's in-house prototype CoC became the *Coq* system, made available to a community of users through an electronic forum. Nevertheless, fundamental issues were not neglected: for instance, Gilles Dowek developed a systematic theory of unification and proof search in Type Theory that was to provide the foundation for future versions of *Coq*.

In 1989, *Coq* version 4.10 was distributed with a first mechanism for extracting functional programs (in Caml syntax) from proofs, as designed by Benjamin Werner. There was also a set of tactics that provided a certain degree of automatization and a small library of developments about mathematics and computer science—the dawn of a new era. Thierry Coquand took a teaching position in Gothenburg, Christine Paulin-Mohring joined the École Normale Supérieure in Lyon, and the *Coq* team carried on its research between the two sites of Lyon and Rocquencourt. At the same time, a new project called Cristal took over the research around functional programming and the ML language. In Rocquencourt, Chet Murthy, who had just finished his PhD in the NuPRL team on the constructive interpretation of proofs in classical logic, brought his own contribution to the development of a more complex architecture for *Coq* version 5.8. An international effort was organized within the European funded Basic Research Action "Logical Frameworks," followed three years later by its successor "Types." Several teams were combining their efforts around the design of proof assistants in a stimulating emulation: *Coq* was one of them of course, but so were LEGO, developed by Randy Pollack in Edinburgh, Isabelle, developed by Larry Paulson in Cambridge and later by Tobias Nipkow in Munich, Alf, developed by the Gothenburg team, and so on.

In 1991, *Coq* V5.6 provided a uniform language for describing mathematics (the Gallina "vernacular"), primitive inductive types, program extraction from proofs, and a graphical user interface. *Coq* was then an effectively usable system, thus making it possible to start fruitful industrial collaborations, most notably with CNET and Dassault-Aviation. This first generation of users outside academia was an incentive to develop a tutorial and reference manual, even if the art of *Coq* was still rather mysterious to newcomers. For *Coq* remained a vehicle for research ideas and a playground for experiments. In Sophia Antipolis, Yves Bertot reconverted the Centaur effort to provide structure manipulation in an interface CTCoq that supported the interactive construction of proofs using an original methodology of "proof-by-pointing," where the user runs a collection of tactics by invoking relevant ones through mouse clicks. In Lyon, Catherine Parent showed in her thesis how the problem of extracting programs from proofs could be inverted into the problem of using invariant-decorated programs as skeletons of their own correctness proof. In Bordeaux, Pierre Castéran showed that this technology could be used to construct certified libraries of algorithms in the continuation semantics style.

Back in Lyon, Eduardo Giménez showed in his thesis how the framework of inductive types that defined hereditarily finite structures could be extended to a framework of co-inductive types that could be used to axiomatize potentially infinite structures. As a corollary, he could develop proofs about protocols operating on data streams, thus opening the way to applications in telecommunications.

In Rocquencourt, Samuel Boutin showed in his thesis how to implement reflective reasoning in *Coq*, with a notable application in the automatization of tedious proofs based on algebraic rewriting. His *Ring* tactic can be used to simplify polynomial expressions and thus to make implicit the usual algebraic manipulations of arithmetic expressions. Other decision procedures contributed to improving the extent of automatic reasoning in *Coq* significantly: *Omega* in the domain of Presburger arithmetic (Pierre Crégut at CNET-Lannion), *Tauto* and *Intuition* in the propositional domain (César Muñoz in Rocquencourt), *Linear* for the predicate calculus without contraction (Jean-Christophe Filliâtre in Lyon). Amokrane Saïbi showed that a notion of subtype with inheritance and implicit coercions could be used to develop modular proofs in universal algebra and, most notably, to express elegantly the main notions in category theory.

In November 1996, *Coq* V6.1 was released with all the theoretical advances mentioned above, but also with a number of technical innovations that were crucial for improving its efficiency, notably with the reduction machinery contributed by Bruno Barras, and with advanced tactics for the manipulation of inductive definitions contributed by Christina Cornes. A proof translator to natural language (English and French) contributed by Yann Coscoy could be used to write in a readable manner the proof terms that had been constructed by the tactics. This was an important advantage against competitor proof systems that did not construct explicit proofs, since it allowed auditing of the formal certifications.

In the domain of program certification, J.-C. Filliâtre showed in his thesis in 1999 how to implement proofs on imperative programs in *Coq*. He proposed to renew the approach based on Floyd–Hoare–Dijkstra assertions on imperative programs, by regarding these programs as notation for the functional expressions obtained through their denotational semantics. The relevance of *Coq*'s two-level architecture was confirmed by the certification of the CoC verifier that could be extracted from a *Coq* formalization of the meta-theory of the Calculus of Constructions, which was contributed by Bruno Barras—a technical *tour de force* but also quite a leap forward for the safety of formal methods. Taking his inspiration from Objective Caml's module system, Judicaël Courant outlined the foundations of a modular language for developing mathematics, paving the way for the reuse of libraries and the development of large-scale certified software.

The creation of the company Trusted Logic, specialized in the certification of smart-card-based system using technologies adapted from the Caml and

Coq teams, confirmed the relevance of their research. A variety of applicative projects were started.

The *Coq* system was then completely redesigned, resulting in version 7 based on a functional kernel, the main architects being Jean-Christophe Filliâtre, Hugo Herbelin, and Bruno Barras. A new language for tactics was designed by David Delahaye, thus providing a high-level language to program complex proof strategies. Micaela Mayero addressed the axiomatization of real numbers, with the goal of supporting the certification of numerical algorithms. Meanwhile, Yves Bertot recast the ideas of CtCoq in a sophisticated graphical interface PCoq, developed in Java.

In 2002, four years after Judicaël Courant's thesis, Jacek Chrząszcz managed to integrate a module and functor system analogous to that of Caml. With its smooth integration in the theory development environment, this extension considerably improved the genericity of libraries. Pierre Letouzey proposed a new algorithm for the extraction of programs from proofs that took into account the whole *Coq* language, modules included.

On the application side, *Coq* had become robust enough to be usable as a low-level language for specific tools dedicated to program proofs. This is the case for the CALIFE platform for the modeling and verification of timed automata, the *Why* tool for the proof of imperative programs, or the *Krakatoa* tool for the certification of Java applets, which was developed in the VERIFICARD European project. These tools use the *Coq* language to establish properties of the models and whenever the proof obligations are too complex for automatic tools.

After a three-year effort, Trusted Logic succeeded in the formal modeling of the whole execution environment for the JavaCard language. This work on security was awarded the EAL7 certification level (the highest level in the so-called common criteria). This formal development required 121000 lines of *Coq* development in 278 modules.

*Coq* is also used to develop libraries of advanced mathematical theorems in both constructive and classical form. The domain of classical mathematics required restrictions to the logical language of *Coq* in order to remain consistent with some of the axioms that are naturally used by mathematicians.

At the end of 2003, after a major redesign of the input syntax, the version 8.0 wasreleased—this is the version that is used in *Coq'Art*.

A glance at the table of contents of the contributions from the *Coq* user community, at the address `http://coq.inria.fr/contribs/summary.html`, should convince the reader of the rich variety of mathematical developments that are now available in *Coq*. The development team followed Boyer and Moore's requirement to keep adapting these libraries with the successive releases of the system, and when necessary, proposed tools to automatically convert the proof scripts—an insurance for the users that their developments will not become obsolete when a new version comes along. Many of these libraries were developed by users outside the development team, often abroad, sometimes in industrial teams. We can only admire the tenacity of this user community to complete very complex formal developments, using a *Coq* system that was always relatively experimental and, until now, without the support of a comprehensive and progressive user manual.

With Coq'Art, this need is now fulfilled. Yves Bertot and Pierre Castéran have been expert users of *Coq* in its various versions for many years. They are also "customers," standing outside the development team, and in this respect they are less tempted to sweep under the rug some of the "well-known" quirks that an insider would rather not discuss. Nor are they tempted to prematurely announce solutions that are still in a preliminary stage—all their examples can be verified in the current release. Their work presents a progressive introduction to all the functionalities of the system. This near exhaustiveness has a price in the sheer size of their work. Beginners should not be rebuked; they will be guided in their exploration by difficulty gradings and they should not embark on a complete, cover-to-cover, reading. This work is intended as a reference, which long term users should consult as they encounter new difficulties in their progress when using the system. The size of the work is also due to the many good-sized examples, which are scrutinized progressively. The reader will often be happy to review these examples in detail by reproducing them in a face-to-face confrontation with the beast. In fact, we strongly advise users to read Coq'Art only with a computer running a *Coq* session nearby to control the behavior of the system as they read the examples. This work presents the results of almost 30 years of research in formal methods, and the intrinsic complexity of the domain cannot be overlooked—there is a price to pay to become an expert in a system like *Coq*. Conversely, the genesis of Coq'Art over the last three years was a strong incentive to make notions and notation more uniform, to make the proof tools explainable without excessive complexity, to present to users the anomalies or difficulties with error messages that could be understood by non-experts—although we must admit there is still room for improvement. We wish readers good luck in their discovery of a difficult but exciting world—may their efforts be rewarded by the joy of the last QED, an end to weeks and sometimes months of adamant but still unconcluded toil, the final touch that validates the whole enterprise.

November 2003 *Gérard Huet*
*Christine Paulin-Mohring*

http://www.springer.com/978-3-540-20854-9

Interactive Theorem Proving and Program Development
Coq'Art: The Calculus of Inductive Constructions
Bertot, Y.; Castéran, P.
2004, XXV, 472 p. 1 illus., Hardcover
ISBN: 978-3-540-20854-9