Machine-checked computer-aided mathematics

Assia Mahboubi

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

HABILITATION À DIRIGER DES RECHERCHES DE

L'UNIVERSITÉ DE NANTES
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Assia MAHBOUBI

Machine-Checked Computer-Aided Mathematics

Habilitation présentée et soutenue à Nantes, le 5 janvier 2021
Unité de recherche : LS2N CNRS UMR 6004

Rapporteurs avant soutenance :

Sylvie BOLDO, Directrice de Recherche, Inria
Thierry COQUAND, Professeur, University of Gothenburg
Lawrence C. PAULSON, Professeur, University of Cambridge

Composition du Jury :

Président :	Yves BERTOT	Directeur de Recherche, Inria
Examinateurs :	Sylvie BOLDO	Directrice de Recherche, Inria
	Thierry COQUAND	Professeur, University of Gothenburg
	Sébastien GOUËZEL	Directeur de recherche, CNRS, Université de Rennes 1
	Florent HIVERT	Professeur, Université de Paris – Saclay
	Nicolas TABAREAU	Directeur de Recherche, Inria

Contents

Résumé en français

Les *assistants de preuve* sont des logiciels conçus pour la réalisation de bibliothèques de mathématiques numérisées. Celles-ci contiennent des définitions, énoncés et preuves tous formalisés dans une variante de logique fixée, de sorte que la vérification de la bonne formation des énoncés, et la correction des preuves, puissent être réduites à un processus mécanique, celui associé au formalisme logique sous-jacent. Le *noyau* de l'assistant de preuve est le composant du logiciel qui réalise cette vérification, tandis que l'assistant de preuve à proprement parler implémente un ensemble de techniques d'automatisation qui permettent à ses utilisateurs de mener à bien la formalisation *en pratique* de définitions et de théories mathématiques arbitrairement sophistiquées.

À ce jour, les assistants de preuves ont principalement été utilisés dans un contexte de vérification de programmes: compilation, programmation système, sécurité, etc. Le théorème à formaliser est alors celui qui spécifie le comportement attendu du programme. Cette spécification peut porter sur des aspects variés du comportement des programmes: correction fonctionnelle, gestion de la mémoire, arithmétique des ordinateurs, etc. Néanmoins, les assistants de preuve ont récemment attiré l'attention d'un nombre croissant de chercheurs en mathématiques, motivés par la vérification de preuves complexes, et non nécessairement calculatoires. La formalisation de ces dernières exige de l'assistant de preuve des formes d'automatisation essentiellement différentes de celles qui servent les besoins de la preuve de programme au sens large. Répondre à ces besoins, pour faciliter la vérification formelle de mathématiques à grande échelle, est un sujet de recherche actuel, qui est le cadre dans lequel se placent les travaux présentés ici.

Ce mémoire présente une synthèse de trois contributions principales à la vérification formelle de théories mathématiques en théorie des types dépendants.

La première de ces contributions porte sur la réalisation d'une bibliothèque de

mathématiques formalisées couvrant les résultats classiques d'algèbre de niveau licence, ainsi que des pans plus avancés de théorie des groupes finis. Ces derniers culminent avec une formalisation du théorème de l'ordre impair (Feit-Thompson, 1963). Il s'agit ici de mettre en perspective les différents ingrédients, de natures variées, qui permettent d'obtenir un corpus ample, cohérent, lisible, réutilisable, et facile à maintenir dans la durée. Ces ingrédients couvrent la conception d'un langage de tactiques, l'utilisation de techniques avancées d'inférence et d'unification, aussi bien que des techniques de formalisation, par exemple celles qui exploitent des schémas de réflexion.

La deuxième contribution porte sur la vérification formelle du théorème d'Apéry, qui établit l'irrationalité de la constante $\zeta(3)$. Au delà de l'intérêt intrinsèque de la vérification formelle ce résultat, il s'agit ici de discuter la coopération de systèmes de preuves formelles avec des systèmes de calcul formel. La preuve vérifiée est en effet une preuve par calcul symbolique, dans laquelle des opérateurs de récurrence jouant un rôle crucial sont calculés par une bibliothèque de calcul formel. Les algorithmes qui produisent ces opérateurs fournissent également des données auxiliaires, des certificats, qui permettent une vérification efficace *a posteriori* de la correction des résultats produits. On discute ici la mise en œuvre de cette coopération, ainsi que ses limites.

La troisième et dernière contribution porte sur la vérification formelle d'algorithmes de calcul numérique *rigoureux*, en particulier de quadratures réelles. Les algorithmes numériques dits rigoureux analysent précisément leur sources d'erreur (d'arrondi et de méthode), et calculent des bornes d'erreur explicites sur leurs résultats. Néanmoins, les conditions sous lesquelles ces calculs d'erreur sont corrects, typiquement une classe de régularité, sont difficiles à tester statiquement sur les entrées fournies par l'utilisateur, ce qui est une source d'erreurs difficiles à détecter. Une implémentation formellement vérifiée permet de s'affranchir de ces erreurs. On discute ici deux approches d'implémentation de quadrature rigoureuse et formellement vérifiée, pour des fonctions réelles et des intégrales propres et impropres.

Remerciements

Je voudrais remercier ici les membres de mon jury: c'est un immense honneur pour moi que chacun d'entre eux ait accepté d'en faire partie. Je remercie tout particulièrement Sylvie Boldo, Thierry Coquand et Larry Paulson pour avoir accepté de rapporter ce manuscrit et pour leur nombreuses remarques et suggestions. Un immense merci aussi à Yves Bertot pour avoir assuré la présidence de ce jury dans le cadre acrobatique imposé par le contexte sanitaire actuel. Sébastien Gouëzel, Florent Hivert et Nicolas Tabareau ont bravé ces circonstances particulières pour m'offrir des conditions de soutenance exceptionnelles pour l'époque : je leur en suis particulièrement reconnaissante.

Je voudrais également remercier Nicolas pour son accueil chaleureux dans l'équipe Gallinette, et à Nantes. Merci à tous les membres de cette équipe pour l'ambiance qui y règne et pour tout ce qu'ils m'ont appris depuis mon arrivée. Les travaux présentés dans ce mémoire doivent beaucoup aux post-doctorants, doctorants, et stagiaires avec lesquels j'ai eu la chance de travailler. Un grand merci à eux tous. J'ai bénéficié de conditions de travail exceptionnelles dans les équipes et laboratoires que j'ai fréquentés ces dernières années: Gallinette, Specfun, Typical, Logical, Marelle, LS2N, LIX, Centre Commun Inria – Microsoft Research. Merci à tous les collègues scientifiques et administratifs qui y ont contribué, ainsi qu'à mes co-auteurs, compagnons de route, de RER B ou de pause café pour tous ces bons moments à faire des mathématiques (ou de l'informatique?) ensemble.

Un immense merci à tous les scientifiques qui ont contribué d'une façon ou une autre à faire exister l'assistant de preuve Coq tel qu'il est aujourd'hui.

Merci à Georges Gonthier pour sa gentillesse, pour ses encouragements, et pour la générosité avec laquelle il partage sa culture et ses idées.

Un merci tout particulier à Claude Jard pour son soutien indéfectible tout au long de cette procédure de soutenance. Merci à Anne-Claire Binetruy et à Annie

Boilot pour leur aide si précieuse. Merci à Charlery Vilar pour la logistique de la soutenance en visio-conférence, qui s'est déroulée dans des conditions optimales.

Merci Erwan. Merci Ines.

Chapter 1

From type theory to machine-checked mathematics

Proof assistants, also called interactive theorem provers, are software tools dedicated to the design of digital libraries of formalized mathematics, which can be *machine-checked* automatically. So far mostly used by researchers in computer science, they have recently attracted the attention of an ever increasing number of researchers in pure mathematics. This trend is happening now because of the successful formalization of recent and major mathematical results, but also because it is becoming clearer and clearer that formalizing mathematics with proof assistants can create novel mathematics.

This introductory chapter aims at putting my recent research contributions in context, while remaining accessible to a non-expert audience.

1.1 Machine-checked mathematics

1.1.1 Computers and proofs

Since the last half of the 20th century, computers have dramatically changed the face of research in mathematics.

It has obviously changed the way people communicate. Researchers in mathematics nowadays seldom exchange snail mails, as Serre and Grothendieck [53, 166] did not so long ago, but electronic messages, like emails or chat group conversions– see for instance the correspondence of the two mathematicians Cédric Villani and Clément Mouhot, reprinted in part in Villani's novel *Birth of a Theorem* [175]. They write research blog posts, they typeset their own articles using scientific doc-

ument preparation systems, and they access them via the Internet –and less and less often so by borrowing a volume at the library.

But this revolution goes beyond communication media. From experimentation to proofs, there is a tremendous momentum right now for computer-aided experiment and guesswork in mathematics. Groundbreaking conjectures were only made accessible by resorting to superhuman computational power, e.g., Mandelbrot's fertile observations in fractal geometry [129], or Birch and Swinnerton-Dyer's conjecture, for which the Clay Institute advertises a $1 million bounty [178]. In fact, they have also revolutionized the essence of what a proof is. Starting with the Four Color theorem, about the coloration of planar maps [11], the list of computer-assisted proofs of major mathematical results has grown steadily, covering a broad range of mathematical fields: combinatorics [113], optimization [90], number theory [30, 96], dynamical systems [169], etc. In turn, the advent of mathematics produced by computer has sparked new forms of mathematics, including computer algebra or numerical analysis.

What makes a candidate proof become a theorem is usually a blend of a social process and of a scrutinizing: people give talks explaining their new ideas to their peers, they eventually submit papers to journals, and these submissions are carefully reviewed by anonymous colleagues. In the vast majority of cases, everything goes fine and only minor errors persist in published texts, that can easily be fixed by the target audience of specialists. But sometimes proofs are published that are truly incomplete or essentially incorrect, and nonetheless believed to be true for a while. There are notorious long sagas of trials and errors like the the Four Colour Theorem [79] or Hilbert's 16th problem [103] – although the latter example is an emblematic example of how fertile false proofs can turn out to be. It is indeed in some cases very difficult to find reviewers who are at the same time: competent enough in all the areas of expertise required, focused enough to detect all the potential flaws, and insensitive to subjective information like their personal relation to the author or his/her reputation in the community. On this topic, the interested reader may watch V. Voevodsky explaining his misfortunes in a popular science talk at the Institute of Advanced Study, Princeton (U.S.A.)[1], and read W. Thurston's reflections [167] on the process of mathematics, written 25 years ago but still current.

Mathematics produced by computer are even more challenging for the traditional model of peer-reviewing, as they require assessing the correctness of the output of complex programs against non-trivial specifications. In the history of mathematical ideas, there is an abundance of example of fertile mistakes and shortcomings: excessive rigor in the mathematical discourse sows the seeds of vacuity. René Thom

[1] http://video.ias.edu/node/6395

even declares that: "Truth is not limited by falsity, but by insignificance." [146]. Is this also the case for bugged computational proofs? Probably much less so. And assessing the absolute correctness of computer programs is a notorious difficult task. Actually, *formal methods* refer to a broad and diverse range of techniques in software and hardware engineering, as well as an extremely active research area in computer science, dedicated to the rigorous assessment of the reliability of software. Yet no explicit policy exists today for auditing software that produce proof steps in submitted papers, even in high-profile mathematical journals. In a number of cases, reproducibility is not even enforced.

But let us temporarily leave aside the possible particularities of computer-produced mathematics. In principle, the whole mathematical literature could be expressed in a non-ambiguous formal language, like set-theory and first-order logic, and proofs could be detailed in an exhaustive manner, making verification (boring to death but) trivial and absolutely objective. Of course, this is not the way people communicate mathematics in practice. The language of mathematics is made of a complex apparatus of abstractions, ellipsis and notations which requires some culture, and years of training to get acquainted with, but which also just makes the communication of ideas possible. Providing all the seemingly missing details to an audience of specialists would not only be utterly pedantic, but in fact it would soon obfuscate the discourse completely, sterilize creativity and miss the point of what makes mathematics beautiful. The famous group of French mathematicians N. Bourbaki make it explicit in *The Architecture of Mathematics* [37]: "*What the axiomatic method sets as its essential aim is exactly that which logical formalism by itself cannot provide, namely the profound intelligibility of mathematics*". Clarifying the abstract structures, which capture the deep similarities shared by seemingly extremely different objects is what it is about. "*To lay down the rules of this language, to set up its vocabulary and to clarify its syntax, all that is indeed extremely useful; indeed this constitutes one aspect of the axiomatic method, the one that can properly be called logical formalism (or "logistics" as it is sometimes called). But we emphasize that it is but one aspect of this method, indeed the least interesting one.*" [37]. To summarize, a verbose expansion of all the definitions and arguments of a given proof, down to the initial actions of a logical foundation, would both be too boring a task and fail to help understanding its content and checking its correctness.

Except that one can today take benefit from the computing skills of machines. The idea of building instruments in order to mechanize the process of verifying deductions is certainly an old one (see for instance G. Leibniz' Calculus Raciocinator in his 1666 doctoral thesis, or W. S. Jevons' Logic Piano built in 1869). But the advent of computers, of modern logic and of the theory of programming languages have turned it into concrete tools that can help for real. *Proof assistants*

provide today an environment for defining mathematical objects, their properties and the associated candidate proofs in a computer formal language well suited to this purpose.

1.1.2 Proof assistants

Many different proof assistants are in use today, just like there are many programming languages, computer algebra systems or document preparation systems. Proof assistants are designed to write and check logic- and computer-based formal proofs, and they usually feature three main ingredients: *an expressive specification language*, used to write formal statements; a distinct programming language for producing candidate formal proofs; a *proof checker*.

A major choice in the design of a proof assistant is the logical foundation used to fix the formal language of mathematics: although most mathematicians are used to set theory and first-order logic, there are many more possible options, some being better-suited to the formalization of mathematical concepts *in practice*. The majority of available proof assistants are actually based on an instance of *type theory*, instead of set theory, and allow quantification on arbitrary objects and functions, instead of being limited to first-order logic. The grammatical constructions and rules of this language are rigid enough so that a program called the *kernel* of the proof assistant can check the correctness of proofs written by the user, by the means of purely mechanical automated process. The kernel is really the cornerstone of a proof assistant: trust in this single piece of code extends to all the proofs it validates. Formalizing mathematics with a proof assistant means forging *by hand* a formal description of the mathematical definitions, theorem statements and candidate proofs: it is worth insisting at this point that only the verification of the latter is automated, and not its discovery. Therefore, formalizing mathematics is about bridging the gap between the very low level language that allows for this routine verification and the much higher-level language that make mathematics intelligible to humans. This gap is akin to the distance between the language in which human beings write programs and the patterns of bits corresponding to the physical commands executed by the processor. The bulk of proof assistants is thus to provide tools that help with this matter. The code implementing these tools is carefully separated from the one of the kernel, so that the trusted base of code remains clearly identified.

Proof assistants, and more generally formal verification tools, can be classified according to the expressiveness of the logical language available to express specifications, i.e., the formal statement of candidate theorems. This language is tight to the choice of mathematical foundations the proof assistant is based on. Different choices for this specification language result in different skills, and different

natures of success. A less expressive logic offers stronger possibilities to automate proof search, a key feature to bridge the gap between the expectations of users and those of the proof checker. On the other hand, a lesser expressiveness imposes higher piles of encoding when formalizing complex mathematical objects, which can hamper the readability, and thus the reliability, of specifications.

Satisfiability (SAT) automated solvers are not usually considered as proof assistants *stricto sensu* because they are designed to be automated instead of interactive, and because they are tied to the unsophisticated propositional logic. They have yet recently earned their stripes in computer-aided mathematics, by finding spectacular proofs of long-standing Ramsey-like conjectures in combinatorics, notably the Pythagorean triples conjectures [99] and the value of Schur number five [98]. In both these cases, certificates, i.e., compact traces, of these proofs were produced by SAT solvers, stored in files of several terabytes, and validated by formally verified checkers [59]. These problems are however arguably quite specific, in that they can be reduced to a bounded, albeit large, quantification: SAT solvers solve them by clever brute force. Beyond finitary problems, the state of the art in formalized libraries of real and complex analysis has been developed using provers based on higher-order logic (HOL). These systems offer unsurpassed *automated proof-search facilities*; users can rely on advanced formal-proof-producing decision procedures and heuristics to close mundane proof obligations [32]. However, the specification language of these provers does not allow a first-class quantification over the instances of a given algebraic structure: extra-logical type classes mechanisms alleviate this issue [101]. But some users report that this limitation can become blocking: see, e.g., Gouëzel's note on the definition of the Gromov-Hausdorff distance between compact metric spaces.[2].

Modern topics in advanced algebra and geometry typically involve complex edifices of sophisticated algebraic structures and constructions. Formalizing in a proof assistant these definitions and their properties calls for a radically different nature of automation, one concerned with the inference of the implicit content carried by notations, and by the meaning of linguistic conventions in the paper literature [125]. Dependent type theory, and more precisely a variant thereof called the Calculus of Inductive Constructions (CIC) [55, 56], provides an extension of higher-order logic that has proved better-suited to a faithful, first-class, representation of hierarchies of *mathematical structures*. The Calculus of Inductive Constructions, and its implementation in the Coq proof assistant, is the main vehicle of the research described in the present memoir.

[2] https://www.math.sciences.univ-nantes.fr/~gouezel/ (accessed May 1st, 2020).

1.1.3 Trusting formally verified mathematics

What do we trust when we trust in machine-checked formal proofs, developed and validated with a given proof assistant? Formal proofs today have become part of the arsenal of formal methods used to stamp code with codified levels of trust [3]. For instance, machine checked proofs have recently gained momentum in security related applications [22, 70, 10]: in this nature of applications it may be in the interest of malicious users to forge validated but incorrect proofs, so as to compromise critical components of a system. But even in a less hostile context, it is worth taking in consideration the ingredients at stake, when we are writing proofs under the guidance and control of a proof assistant.

Trust certainly has to be put in the foundations underlying the proof assistant under consideration. In most cases, basing such a tool on an inconsistent proof system defeats the whole point of formalization, as any statement could be derived *ad absurdum*. The case of proof assistants based on dependent type theory poses some specific issues in that respect. This meta-theory is notoriously subtle, and a combination of seemingly innocuous extra ingredients can make the whole edifice collapse [45, 141]. The construction of models which justify extensions of a core dependent type theory can also prove a delicate exercise, although syntactic techniques have recently helped clarification [36]. Actually, consistency is not the only desirable property of the formalism: others facts like subject reduction or the decidability of type-checking have an important impact as well on the practice of formalization. Last, there is possibly a non-trivial act of faith in the belief that the type theory actually implemented in the proof assistant, coincides precisely with the formalism studied on paper: beyond the possible bugs, some implementation choices or optimizations can actually affect the formalism.

The proof checker of a proof assistant is the obvious critical piece of the trusted base of code of a proof assistant and a fertile line of research has been devoted to self- or cross- verification of proof checkers [92, 20]. The reliability of proof checking was actually a central motivation of Robin Milner in his design and implementation of the Logic of Computable Functions (LCF), a variant of a λ-calculus strongly inspired by Scott and Strachey. The so-called *LCF approach* [85] refers today to the design of proof checkers along three main ingredients[4]: a special abstract type for theorems; constructors of this abstract type representing the inference rules of the logical system; a strongly-typed high-level language for the implementation of the proof assistant. Proof assistants from the HOL family follow this approach to the letter. As for the so-called *de Bruijn criterion*, as coined by Henk Barendregt and

[3] See for instance Bertot et al.'s position paper on the use of Coq for Common Criteria Evaluations.

[4] This trio is borrowed from a talk by Harrison.

Freek Wiedijk [19] after the architect of the pioneer AUTOMATH system [136], it requires the possibility of independent proof checking by a small program: proof assistants based on CIC, like Coq, Agda, Lean, Matita fulfill this condition by exporting proof objects, that can in principle be verified by an independent checker.

A proof assistant however is much more than a proof checker, and, more often than not, formalization would not be possible at all without the assistance of complex elaboration procedures, which transforms the input from the user into a complete term of the formalism, ready to be machine-checked. Without strong elaboration features, formal statements quickly become too verbose to be readable (and one becomes exposed to the threat of formalization choices exposed in the next paragraph). Therefore, inevitably, what the user sees is not what the proof checker gets. Elaboration, although invisible to the user, nonetheless involves multiple ingredients: parsing, type inference, unification, insertion of coercions, etc. Pollack inconsistency is an example of what can go wrong [177]. The extension of type inference with so-called type class mechanisms [89, 160] raises issues of a similar nature about the resolution of advanced notation devices, and with the corresponding inferred meaning. An informative interaction with the users is key to mitigate the threat of uncontrolled elaboration, and the meta-properties of the underlying formalism play an important role here.

Last but not least, formalization choices matter. Users can get their formal definitions wrong, that is, they can devise formal definitions that do not coincide with the mathematical objects they have in mind. And proof assistants are not entitled to check definitions, which often requires several iterations to stabilize. This threat is rather absent from synthetic mathematics, e.g. synthetic topology [71] or synthetic homotopy theory [147], but they become more significant in other fields, in which formal definitions are made of higher piles of intermediate objects. For some extreme cases of sophisticated definitions, it might even be the case that constructing formally an instance of a given abstraction represents a true challenge. The authors of a formal definition of Schölze's perfectoid spaces, a *tour de force* conducted in the Lean proof assistant, report and discuss this sort of difficulty [41].

Addressing these issues requires theoretical work, both on the foundations and on the algorithms implemented by proof assistants, but it also requires improving the design and the implementation of the inspection and visualization tools offered by proof assistants: all these topics remains as of today research topics of their own.

1.1.4 Computation inside logic

The two systems considered as the first modern proof assistants were independently designed in the late 60s. The AUTOMATH system, by Nicolaas G. de

Bruijn [136] was based on a variant of λ-calculus with dependent types and its claim to fame was the formal verification, by Jutting, of Landau's *Foundations of Analysis*. The Mizar system, founded by Andrzej Trybulec [168], is based on Tarski-Grothendick's variant of set theory, and uses of soft type system to help codifying formal statements. Still in use today, it has been used to produce one of the largest existing libraries of formalized mathematics [18].

Although these two pioneers have been motivated by the formalization of mathematical theories, the successors of these systems have so far mostly attracted users with a computer science background, concerned with obtaining the highest possible confidence in the behavior of *programs*. Describing accurately the properties of the output of an algorithm, and the behavior of a program implementing this algorithm, and then, proving that these specifications are fulfilled indeed constitutes a notoriously difficult task, which can involve arbitrarily advanced mathematics. We provide below a brief overview of the different approaches to machine-assisted program verification, biased towards computational mathematics and in particular, computer algebra.

A proof assistant based on type-theoretic foundations offers the possibility to implement an algorithm directly as a λ-term, i.e. as an object in the logic. The program can subsequently become the subject of a formalized statement, candidate theorem expressing a desired correctness property. The example given in Listing 1.1 defines a function `cormen_lup` which compute an LUP decomposition of its argument, a square matrix with coefficients in a field, according to the classic recursive algorithm explained for instance in Cormen et al.'s reference book [57]. The output of the algorithm on an input matrix A is a triple $((P, L), U)$ of square matrices, such that P is a permutation matrix, L is an unipotent lower matrix and U is an upper triangular matrix. The term `cormen_lup` is a program, written in Gallina, the programming language integrated to Coq's logical foundations. The properties ensuring the correctness of the algorithm are respectively formalized as the statements of lemmas `cormen_lup_perm`, `cormen_lup_p_correct`, `cormen_lup_lower`, `cormen_lup_upper`.

The kernel of Coq event implements optimized reduction strategies, notably a virtual machine [86], for evaluating this nature of programs efficiently, when they operate on appropriate data structures (unlike the present example, which is not meant to be executed in practice).

```
Section CormenLUP.

Variable F : fieldType.
Implicit Types A : 'M[F]_n.+1.

Fixpoint cormen_lup {n} :=
```

```
  match n return let M := 'M[F]_n.+1 in M -> M * M * M with
  | 0 => fun A => (1, 1, A)
  | _.+1 => fun A =>
    let k := odflt 0 [pick k | A k 0 != 0] in
    let A1 : 'M_(1 + _) := xrow 0 k A in
    let P1 : 'M_(1 + _) := tperm_mx 0 k in
    let Schur := ((A k 0)^-1 *: dlsubmx A1) *m ursubmx A1 in
    let: (P2, L2, U2) := cormen_lup (drsubmx A1 - Schur) in
    let P := block_mx 1 0 0 P2 *m P1 in
    let L := block_mx 1 0
             ((A k 0)^-1 *: (P2 *m dlsubmx A1)) L2 in
    let U := block_mx (ulsubmx A1) (ursubmx A1) 0 U2 in
    (P, L, U)
  end.

(* First element of the triple is a permutation matrix *)
Lemma cormen_lup_perm n A :
     is_perm_mx (cormen_lup A).1.1.
Proof. ... Qed.

(* Identity relating input and outputs: the input A permuted
   by P is equal to the product L * U *)
Lemma cormen_lup_correct n A :
  let: (P, L, U) := cormen_lup A in P * A = L * U.
Proof. (...) Qed.

(* Second element of the triple is unimodular *)
Lemma cormen_lup_detL n A :
  \det (cormen_lup A).1.2 = 1.
Proof. (...) Qed.

(* Second element of the triple is lower triangular,
  with 1s one the diagonal. *)
Lemma cormen_lup_lower n A (i j : 'I_n.+1) :
  i <= j -> (cormen_lup A).1.2 i j = (i == j)%:R.
Proof. (...) Qed.

(* Third element of the triple is upper triangular *)
Lemma cormen_lup_upper n A (i j : 'I_n.+1) :
  j < i -> (cormen_lup A).2 i j = 0 :> F.
Proof. (...) Qed.

End CormenLUP.
```

Listing 1.1: LUP Decomposition of a matrix and the related correctness proofs

Computations carried inside the logic are as trustworthy as the proofs validated by the proof checker of the proof assistant; this is the approach adopted both in the verified proof of the four colour theorem [79], using Coq, and in the verified proof

of the Kepler conjecture, using HOL-Light and Isabelle/HOL [90]. Proof assistant thus provide an appealing solution to the problem of assessing computer-produced mathematics.

Of course, computing inside the logic incurs a possibly huge efficiency cost. A fruitful approach thus consists, whenever possible, in relying on external oracles to produce *certificates*, later checked inside the prover [94]. This way, exploration of a large search space can be performed by efficient, unverified oracles, before a verified checking procedure achieves the final formal validation step. Primality (dis)proving is an obvious application of this approach, and existing formal-proof-producing primality tests are efficient enough to prove large primes [88], although unfortunately not of the size needed in Helfgott's proof of the Ternary Goldbach conjecture [97].

In this perspective, several attempts have also been carried to build *bridges* between computer algebra systems, and proof assistants, so as to ease their cooperation, e.g., Coq and Maple [61], HOL-Light and several systems [109], Lean and Mathematica [119]. These bridges are however very fragile, as both ends are moving targets. Moreover, interaction is hampered by the lack of formal semantics of computer algebra systems, as well as by the lack of standards for sharing data-structures such as polynomial expressions. Nonetheless, the verification [47] of Apéry's proof of the irrationality of $\zeta(3)$, described in Chapter 3, illustrates the huge potential benefits of a smoother interaction between trusted and non-trusted computer-algebra programs.

The most successful implementations of computer algebra inside proof assistants actually rather serve the purpose of enhancing the *automation* of the provers, with (semi-)decision procedures, e.g., normalization of algebraic expressions [87], of linear and non-linear real arithmetics [131, 28, 42], automated proofs of asymptotic behaviors [68], and of bounds on special functions [159, 130]. But the average efficiency is dreadful, compared to what computer algebra systems offer. These procedures are in fact designed to solve the possibly numerous yet small problems generated by proof obligations, instead of very large instances.

1.1.5 Program verification using proof assistants

The optimization allowance when computing inside logic remains limited. Computing inside logic is fortunately not the only option to perform verified computations using proof assistants, as most systems provide *code generation* features, sometimes also called *extraction*. These features make it possible to translate a program represented inside the logic, and possibly endowed with a formal proof of correctness, into another program, this time written in a realistic programming

language, and meant to be executed. Compared to the execution of programs inside logic, this approach pulls a larger environment into the trusted base of code. On the other hand, such a compromise appears as a necessary trade-off in order to verify computer-produced mathematics at large. For instance, computer algebra systems owe a large part of their stunning efficiency to their arithmetic and linear algebra core. And an efficient implementation of such a core is far more demanding than what internal computations to the logic alone can offer.

Code generation features in proof assistant have been quite extensively used. The Coq proof assistant implements a variant of realizability, which can produce pure, functional Ocaml programs, from programs written in the pure, functional programming language embarked in Coq's logic, and even from axiom-free existential Coq proofs. For example, the CompCert verified compiler [117] or the Verasco verified static analyzer [108] generate their executable code this way. A similar feature is used to produce, from Isabelle/HOL functional code, a verified factorization algorithm for polynomials, based on LLL basis reduction [62], with comparable performance to the corresponding routine in Wolfram/Mathematica, as well as a verified, rigorous solver of Ordinary Differential Equations [104].

Frameworks like PVS2C [58], for the PVS prover, or Imperative/HOL [116], for the Isabelle/HOL prover make it possible to refine *pure data structures* into *imperative code*. The latter framework was used to generate a verified SAT solver whose performances come close to the ones of Mini-SAT [77]. The Lean prover, implemented in C++, can even be bootstrapped from its meta-programming and code generation features [171, 69]. In all the above examples however, code generation is not formally verified. The CakeML ecosystem, by contrast, can produce *stateful ML programs*, together with proofs of correctness, from *monadic functions in HOL*, and then generate verified machine code, using a verified compiler back-end [100].

The efficiency of state-of-the-art arithmetic packages, such as the GNU Multiple Precision Arithmetic Library (GMP), and linear algebra packages, such as the Basic Linear Algebra Subprograms library (BLAS), comes from fine-tuned, *in-place operations* on *low-level data structures*, e.g., imperative arrays and pointers. Each operation comes in several implementations, each specialized to a different class of entry size. For instance, the schoolbook, quadratic, multiplication algorithm is optimal for numbers shorter than 2000 bits, but beyond, the Toom-Cook family of interpolation-based algorithms is more relevant. The most efficient algorithms in these packages are intricate, and their correctness relies on subtle mathematical tricks, in order to optimize resources. These optimizations are outside the scope of compilation, and require operating by hand on low-level features. The formal verification of arithmetic routines has been motivated by its usage in safety-critical applications, such as cryptography or security of Internet. By definition, it requires

reasoning on the *memory model* of the implementation language, typically a variant of C.

The available verification approaches roughly fall in two camps, which we illustrate on the verification of multi-precision arithmetic functions or libraries. The first approach consists in using a proof assistant to model the programming language constructs, its memory model, and to verify the correctness of a model program. Two decades ago, a proof of the general case of the GMP square root algorithm would take 13,000 lines of code (loc) of Coq [27], when a decade later a Coq proof of about the same size can deal with a binary extended GCD algorithm, as well as the functions it depends on, such as addition, subtraction, and halving [7]. Automation is key to success in this area. Thanks to the strong support for automation provided by the Isabelle/HOL prover, it takes only 2000 lines of code to verify a bignum library programmed in the SPARK fragment of the Ada programming language, using a verification framework that sends goals to the prover. None of these achievements however connects directly proofs to actual, executed machine code. Perhaps the only exception is a verified implementation of arbitrary-precision integer arithmetic, using the HOL4 theorem prover [135], which is turned into x86 machine code using CakeML. But the latter implementation is not geared towards efficiency (e.g., the multiplication is the schoolbook one).

The other approach consists in using a different kind of deductive tools, which emphasizes automation, typically by Satisfiability Modulo Theory (SMT), rather than foundational correctness. Such tools are geared towards the verification of realistic code, that can be included in subsequent applications. For instance, the Mozilla Firefox web browser now uses code from a library for elliptic curve cryptography, developed using the F* verification system [183]. The Why3 verification platform was used to generate a comprehensive verified C library, for arbitrary-precision integer arithmetics, with a comparable efficiency to the architecture-independent mode of GMP [132].

The work presented in the present manuscript mostly belongs to the approach consisting in computing inside logic. However, our closing chapter discusses the limits and perspectives of this choice, in the light of the other approaches we over-viewed in the present introduction.

1.2 Structure of the manuscript

This manuscript provides a synthesis of a selection of work, carried with my collaborators since the completion of my PhD, and published in peer-reviewed venues. At some places, excerpts of these publications are included verbatim. It does not include any new contribution, and is intentionally non-technical: details can be

found in the corresponding peer-reviewed and published papers, which should remain the main reference. For each chapter, a concluding section outlines a few comments and draws some perspectives for future work.

A large part of Section 1.1, in the present chapter, is based on the expository article [121], as well as on discussions with Guillaume Melquiond [5].

Chapter 2 presents two contributions in formalized mathematics. One is a study of quantifier elimination proofs, in particular for real and algebraic closed fields: this work was mostly conducted in collaboration with Cyril Cohen while he was a PhD student under my supervision. The other is about finite group theory: it is the outcome of the 6-year effort of the Mathematical Component team, culminating with a formal proof of the Odd Order Theorem. The contributions here consist in illustrating how various insights from computer science, and more precisely, from the theory of programming language, can be put at the service of formalized mathematics in practice. This chapter is largely based on (a synthesis of) my publications on the topic [51, 50, 120, 81].

Chapter 3 presents a contribution in formally verified computer-produced mathematics, using symbolic methods. It describes a formal verification of Apéry's proof of the irrationality of $\zeta(3)$, based on an interaction between a computer algebra system and a proof assistant. This work was conducted in collaboration with Frédéric Chyzak and Thomas Sibut-Pinote, while Thomas Sibut-Pinote was a PhD student under my supervision. This chapter is largely based on my publication on the topic [47], as well as on a submission currently under revision [124].

Chapter 4 presents a contribution in formally verified computer-produced mathematics, using symbolic-numeric methods. It describes a formally verified routine for computing numerical enclosures, and quadratures in particular, based on interval arithmetic. This work has two parts: the first was conducted in collaboration with Guillaume Melquiond and Thomas Sibut-Pinote, while Thomas Sibut-Pinote was a PhD student under my supervision. The second, which proposes a certificate-based approach for this problem, stems from a collaboration with Florent Bréhard and Damien Pous. This chapter is largely based on my publications on the topic [122, 123, 40].

Finally, Chapter 5 outlines the research program I would like to explore for the coming years. It has benefited from many discussions with a number of colleagues, notably Jasmin Blanchette, Sander Dahmen, Maxime Denès, Guillaume Melquiond and Thierry Priol.

[5] Its content however remains under my sole own responsibility.

Chapter 2

Libraries of formalized mathematics

2.1 The Mathematical Components library

2.1.1 Overview

The Mathematical Components library[1] is a library of formalized mathematics for the Coq proof assistant, geared towards algebra. Its first release, in 2008, mostly included a small collection of files from Georges Gonthier's Four Colour theorem proof [79], and an extension to Coq's tactic language called SSReflect [82]. As of today, most of the content of this initial release has been merged into the standard distribution of the Coq proof assistant, including the corresponding documentation, which has become a chapter of Coq's reference manual. The Mathematical Components library today mostly consists in the general-purpose parts of the libraries written for a formal proof of the Odd Order theorem (see Section 2.3): graduate level finite group theory, matrix algebra, commutative algebra, etc. Since its first release, the library has been continuously maintained and adapted to the evolution of the Coq proof assistant. But it has also been enriched, with new formalized content (e.g. related to ordered structures) and with improved infrastructure (e.g. tooling for algebraic hierarchies).

The corpus of formalized theories to be found in the Mathematical Components library features a uniform programming style, naming and notation policies, and formalization choices. The rationale behind the programming style is chiefly readability and maintainability. These objectives are served by the SSReflect extension of the Coq's tactic language. In particular, this tactic language enhances the support for declarative style, so as to make possible the management of long proofs with numerous proof steps and large contexts, and this in particular when con-

[1] https://github.com/math-comp/math-comp

texts are populated by piles of local definitions rather than by case analysis on inhabitants of sophisticated inductive types. Using this style, the readability of the proof is ensured by the readability of intermediate mathematical statement, rather than by the one of the proof script. This guiding principle is not original, it is for central to the interaction model in Isar [176], for the Isabelle/HOL prover.

As a rule of thumb, the formalization choices adopted in the library favor a faithful behavior in proofs to a syntactical similarity in formal definitions. The library thus includes a significant amount of infrastructure material, e.g., for types with a decidable equality (a special case of h-set), for proof-irrelevant sigma-types, for iterated binary operators [26], etc. Infrastructure typically combines dependent records, type-classes, and notations, with a possible additional support at the level of the tactic language. Inductive-shaped specification lemmas, e.g. using the `reflect` two-constructor inductive type, are also pervasive. More details about these technical aspects are provided in the Mathematical Components reference book[126].

2.1.2 Structures, type inference and notations

One of the main issues in formalization is to model the training a mathematician went through to be able to make sense of a mathematical text: this is crucial keep formal statement intelligible and to tame bureaucracy in formal proofs.

Consider for instance the sentence:

Let R be a commutative ring, the determinant $Det(A)$ of a square matrix $A \in M_n(R)$ verifies the formula:

$$Det(A) = \sum_{\sigma \in S_n} \varepsilon_\sigma \prod_i a_{i,\sigma(i)}.$$

A generic undergraduate student is able to infer without noticing that the Greek letters Σ and Π, notations for iterated binary operations on the domain described in subscript, here apply respectively to the addition and to the multiplication of the ring structure which equips R, that the iteration domain of the product is necessarily iterated for $1 \leq i \leq n$ and that the signature ε_σ should be understood as its embedding in R. Although none of these mundane details is explicitly denoted in the formula, they can all be deduced from the context of the formula. The missing-but-obvious information in such a string of symbols is however necessary for the computer to make sense of the statement. Yet it would not be reasonable to expect the user of the proof assistant to provide it by hand explicitly, nor to decorate excessively the words constituting the formal sentence. She would soon not more see the forest for the trees. Fortunately the training of the reader can

be modeled by implementing and running appropriate algorithms which are able to infer the canonical properties hidden behind standard notations, once these rules and habits have been spelled out completely. Such algorithms have to do with *type inference*. As in usual programming languages, types are labels carrying information like the domain and co-domain loci of functions. Well-formed sentences satisfy constraints on their types, which are prescribed by the rules of the formal language. This helps structuring the sentences and ruling out nonsensical ones, hence facilitating the verification. But not all type annotations need to be provided by the user, as some of them can be retrieved by just enforcing the constraints on the types. This mechanism, called *type inference*, also comes from the theory of programming languages and can be used to reconstruct what the trained reader can read behind the notations, by encoding enough information in the types. This has been extensively used in the libraries of the Odd Order Theorem formal proof to preserve the readability of statements, the modularity of the theories and the ability to superimpose several views on a same object. In the end, the code producing the formula $\sum_{\sigma \in S_n} \varepsilon_\sigma \prod_i a_{\sigma(i),i}$ in the LaTeX typesetting language is:

```
\sum_{\sigma \in S_n}\epsilon_{\sigma}\prod_i a_{\sigma (i),i}
```

when an analogue in Coq would be:

```
Definition det (R : ringType) n (A : 'M[R]_n) : R :=
  \sum_(s : 'S_n) (-1) ^+ s * \prod_i A i (s i).
```

For a more detailed insight into the use of type inference in the formalization of mathematics, the interested reader can refer to Jeremy Avigad's introduction [15].

The Mathematical Components library has pioneered the use of a flavor of unification hints [14], called canonical structures [125], to implement this nature of enhanced type inference in proof assistants based on dependent type theory. In Coq, the unification algorithm can use these hints to infer the value of a dependent tuple (i.e., a dependent record type) from the value of a single field. The solution picked by unification to such a unification problem, otherwise unsolvable, is the one previously stored by the user in the database of hints. The Mathematical Components library features a hierarchy of mathematical structures (and morphisms) [78], built using this mechanism and populated with a large number of instances. But other approaches exist to the implementation of type classes in dependent type theory, usually based on enhancing the inference of implicit arguments [160], rather than on unification hints. They have also been used to build hierarchies of structures, in Coq [161] as well as in the Lean proof assistant [3]. The appropriate design of such hierarchies remains as of today research in progress.

2.1.3 Elementary vocabulary of finite group theory

In the preceding section, we have mentioned how enhanced type *inference* can be used for implementing proof search, inferring for the dumb kernel some bureaucratic information that the human user would like to be kept implicit. Inappropriate formalization choices might however trigger blocking issues if typing *constraints* become too demanding, because types are too fine-grained. The formalization of the elementary vocabulary of finite group theory illustrates this issue, as described in more details in the corresponding publications [83, 120].

In this case, a key insight has been to start with two distinct structure types when crafting the definition of groups: a type of pre-group domain, coined `baseFingroupType`, for pointed finite types equipped with a monoid law and an involutive antimorphism, and a type `finGroupType` of group domain, which refines `baseFingroupType` into a structure for finite types equipped with a group law. The purpose of these two structures is to prescribe common laws to the collections of their inhabitants, even if only some of these collections (satisfying suitable properties) are actual instances of groups. The distinction between group type and pre-group type allows to define most of the notations at the level of subsets: for instance, notation `1` refers both to the neutral element of a group and to its associated singleton set (the trivial group). Note however that the possibility to trigger a reduction in any (sub)term, at any step of a term comparison, makes the interaction between coercions and notations are quite subtle. Crafting robust notations, ones that do not vanish unexpectedly because reduction accidentally erased the corresponding constant, is subtle art. This requires extra care in formal definitions, which are best described in the comments of library `fingroup.v`.

More generally, the distinction between domains and actual groups is crucial to loosen enough the typing constraints under which a formula with group theory notations is well-formed. For instance, for two subset A and B of a pre-group domain $\mathcal{G}$, the set product is defined as:

$$A \cdot B = \{a * b \mid a \in A, b \in B\}$$

and the normalizer set of A is:

$$N(A) = \{x \mid A^x \subset A\} \text{ where } A^x = \{x^{-1}ax \mid a \in A\}.$$

We will also casually write AB for $A \cdot B$. A group of a pre-group domain type $\mathcal{G}$ is thus a finite subset $G \subset \mathcal{G}$ such that $1 \in G$, where 1 is the distinguished point in $\mathcal{G}$, and $G \cdot G \subset G$, otherwise said that G is stable under the monoid law. When the set H is equipped with a group structure, then so is $N(H)$, and this fact is retrieved automatically thanks to the registration of the corresponding canonical instance of group via a unification hint.

Finding the right formal definition for group morphisms required a few iterations. A group morphism between two group domain $\mathcal{G}$ and $\mathcal{H}$ is the data of a subset D of $\mathcal{G}$ and of a function φ from $\mathcal{G}$ to $\mathcal{H}$, such that $\varphi(xy) = \varphi(x)\varphi(y)$ for any $x, y \in D$. As a consequence the *morphic* image of a set $A \subset \mathcal{G}$ under this morphism is the set $\varphi(A \cap D)$ and the morphic pre-image of $B \subset \mathcal{H}$ is $\varphi^{-1}(B) \cap D$.

These choices allow a satisfying formalization of elementary finite group theory, notably including the three group isomorphisms. As elementary as it seems, obtaining an appropriate version of these statements, usable in practice in proofs, was not that easy. A discussion of the formalization of the third isomorphism theorem, as well as on the definition of composition series and the Jordan-Hölder theorem can be found in this survey paper [120].

A central guiding principle in the library is the following: for definitions operating on "sets with a possibly additional structure", like operations on groups (product, quotient, normalized, semi-direct product, etc.), generalize the definitions and notations as much as possible, so that they become as liberal as possible. Then endow the resulting objects with the desired structure, when parameters are themselves equipped with the adequate requirements. For instance, on paper, writing G/H is only allowed when it is known *at the time of writing* that H is normal in G. But incorporating such a requirement in the type would be too painful: the statement of the third isomorphism theorem is an example of this pain. Instead, the Coq term `G / H` is well typed as soon as the two groups involved are subsets of a same group domain, that is `(G, H : set gT)`, for some `(gT: finGroupType)`. But of course it has the expected meaning only when `G` and `H` are actual groups. Let us comment on a last example, the formalization of semi-direct products of two groups of a same group domain $G = N \rtimes H$, which is defined on paper only when N is normal in G, $G = NH$ and $N \cap H = 1$. We first introduce a definition of a *partial* product, which assigns a default value, the singleton $\{1\}$ to the set product NH when N or H is not a group.

```
Variables gT : finGroupType.
Implicit Types A B C : {set gT}.

Definition partial_product A B :=
if A == 1
then B
else if B == 1
     then A
     else if [&& group_set A, group_set B & B \subset 'N(A)]
          then A * B
          else set0.
```

where `[&& _, _ & _]` is a ternary (boolean) conjunction and the `group_set` predicates

tests whether a subset of a group domain is a group or not, by checking for the presence of the neutral element, and for the stability under the law:

```
Definition group_set A := (1 \in A) && (A * A \subset A).
```

Note that the output of `partial_product` always verifies `group_set`. The corresponding formal definition, as a subset of the group domain type, is then the following:

```
Definition semidirect_product A B :=
  if A :&: B \subset 1 then partial_product A B else set0.

Notation "N ><| H" := (semidirect_product N H).
```

Note that `N ><| H` is a subset of the group domain, even when `N` and `H` are both groups: they are coerced to their carrier set before being passed to `partial_product`. The intended way to recover a group structure is to introduce a third *group* `G` and to state and use an equality `N ><| H = G`, in particular via the following lemma:

```
Lemma sdprodP A B (G : {group gT}) :
A ><| B = G ->
[/\ are_groups A B,     (* A and B are group_set *)
    A * B = G,          (* A* B is G as a set    *)
    B \subset 'N(A) &   (* B is a subset of N(A) *)
    A :&: B = 1].       (* intersection of A and B is {1} *)
```

where `[/, _ _, _ & _]` denotes a 4-ary conjunction Note that a distinct variant of semi-direct product allows to forge a product of two groups living in distinct group domains, by providing the defining action explicitly.

2.2 Formalized proofs of quantifier elimination properties

2.2.1 Quantifier elimination for first-order theories

Quantifier elimination is a standard way of reducing the decidability of a first-order theory to the validity of its quantifier-free formulas. Typical proofs go by induction on the structure of a formula, and the crux of the proof is to establish that formulas possibly featuring free variables, but with a single existential quantifier, have a quantifier-free equivalent. This restricted elimination principle is akin to proving that a certain class of constructible sets is stable by projection. Examples of theories enjoying quantifier elimination include linear orders but also Presburger (linear) integer arithmetic.

The first-order field theory of algebraically closed fields and that of real closed fields both also enjoy quantifier elimination. We recall that algebraically closed

fields are fields which split any univariate polynomial with coefficients in the field, at that their first order theory is the collection of first order equational formulas in the language of rings. Real closed fields can be characterized as totally ordered fields whose positive elements are exactly the squares. For instance, real algebraic numbers are constructively equipped with a structure of real closed field. The first order theory of real closed fields thus involves atoms made with the order relation (strict and large) in addition to the equational ones.

Both these quantifier-elimination results are attributed to Tarski [165]. The corresponding geometrical formulation, stating that projections of constructible sets are themselves constructible is known as Chevalley's Constructibility theorem [44].

Proofs of both these results are formalized in the Mathematical Components library[2].

2.2.2 Formalization in dependent type theory

The formal statements of these proof-theoretic results make use of several specific features of the Calculus of Inductive Construction, and would have a very different wording if formalized in, e.g., a proof assistant from the HOL family. In particular, the expressivity of dependent type theory enables a first class status to the statement of a quantifier elimination property, by a single quantification over all the instances of the structure under interest (here closed and real fields respectively), using a dependent record type which bundles the corresponding carrier, data and axioms. The formal statements also involve an inductive type for first-order formula in the language of interest, which reifies the theory, and an interpretation function of such a formula in any instance of the structure.

Consider T a first-order theory on a signature Σ. By definition, a Σ-structure M is a model of T, denoted $M \models T$ if for any (closed) formula $\varphi \in T$, $M \models \varphi$. We say that two (non necessarily closed) formulas φ and ψ are equi-satisfiable in a given model M if for any context e, ($M, e \models \varphi$ if and only if $M, e \models \psi$). We say that a theory T admits quantifier elimination, if for every (non-necessarily closed) formula $\varphi \in T$, there exists $\psi \in T$ such that ψ is quantifier free and for any model M of T, and for any list e of values, $M, e \models \varphi \Leftrightarrow \psi$.

The theory considered here is the collection of first-order formulas in the language of fields with constants in a real (resp. closed) fields. The following `formula` type provides a deep-embedding of this theory:

```
Variable (R : Type).
```

[2]More precisely, the proof for real closed fields is a satellite project in the Mathematical Components github organization.

```
Inductive term : Type :=
| Var of nat (* variables *)
| Const of R (* constants *)
| Add of term & term (* addition *)
| Opp of term (* opposite *)
| Mul of term & term (* product *)
| Inv of term (* inverse *).

Inductive formula : Type :=
| Bool of bool
| Equal of term & term
| And of formula & formula
| Or of formula & formula
| Implies of formula & formula
| Not of formula
| Exists of nat & formula
| Forall of nat & formula.
```

where quantifiers explicitly take as argument the name of the variable they bind. We now define a Coq predicate

```
  holds : ∀F : unitRingType, seq F -> formula F -> Prop
```

The first argument `(F : unitRingType)` of the `hold` predicate is an instance of ring with units, e.g., a field: it is the minimal requirement for being able to interpret the complete signature reified in the abstract syntax tree of formulas. Its second argument `(e : seq F)` is a sequence of terms in the carrier type of field `F` (a hidden coercion projects the field onto its carrier). Its last argument `(f : formula F)` is a formula whose atoms may feature constants interpreted in the carrier type of the field `F` (again, a coercion is inserted). The definition of the `hold` predicate, by induction on the structure of the formula, is such that `(holds F e f)` is $F, e \models f$. It uses an auxiliary evaluation function, for terms, also defined by induction, this time on the structure of a syntactic term `(t : term)`.

Now for any type `(T : Type)`, it is straightforward to test whether a formula `(t : formula T)` is quantifier free: one just tests recursively that `t` does not feature any `Exists` nor `Forall` constructor. This results in a boolean test:

```
Definition qf_form : ∀T :Type, formula T -> bool.
```

Now the Coq theorem we prove is that there exists a transformation:

```
Definition q_elim: ∀F, formula F -> formula F
```

such that the following property holds:

```
Lemma q_elim_wf: ∀F (f : formula F), qf_form (q_elim f).
Lemma q_elimP: ∀F (f : formula F) (e : seq F),
  holds e f <-> holds e (q_elim f)
```

where `F` will be in one case an algebraically closed field, i.e. `(F : closedField)`, and in the other a real closed field, i.e., `(F : realClosedField)`.

More precisely, the hierarchy of structures alluded to in Section 2.1.2 features an interface for fields with a decidable first-order theory, from which the interfaces of real an algebraically closed fields inherit.

2.2.3 Proofs

The formalized proof of quantifier elimination for algebraically closed fields follows a standard reference by Saugata Basu, Richard Pollack and Marie-Françoise Roy [23]. This formal development has been the occasion to expand the theory of polynomial divisibility. But the main contribution, of this work, mostly due to Cyril Cohen, has been to find a clever way to perform the computation involved in the projection argument, and thereby to significantly improve on the classical presentations.

The crux of the projection lemma is actually to construct, from a formula with n free variables and a single existential quantifier, a new, quantifier-free formula which describes a suitable partition of the space F^n, with a system of polynomial equations (or disequations) for each cell. The initial, quantified formula can be seen as the statement of the existence of a solution to a certain system of polynomial constraints in the bound variable, where the coefficients of the univariate polynomials involved in the systems are themselves polynomials in the parameters (i.e., in the free variables). The different cells of the partition roughly correspond to the different combinations of results obtained on testing for nullity quantities obtained from these polynomial coefficients. In order to gradually compute this formula, akin to an execution trace, the program mimics the proof with functions operating of formal terms and written in continuation-passing style. This technique makes the construction of the formula easy to describe, but also smoothens the formal proofs of the correctness lemma `q_elimP`. The technical details are given in the corresponding papers [50, 51].

The proof of quantifier elimination for real closed fields is much more intricate in essence, even for variants which do not attempt to control the complexity of the underlying algorithm. The version of the proof formalized in this work indeed has the same complexity as the one by Tarski, that is, a tower of exponentials of height the number of quantifiers to be eliminated in the total degree of the polynomials involved in the formula. This formal proof uses the same technique based on continuation-passing style to implement quantifier elimination, but this time, the elimination procedure is more sophisticated. Indeed, the description of the cells in the partition is more involved, and it is essentially based on the

study of variants of pseudo-reminder sequences. An additional technicality comes from the constraint of formulating the proof in the language of integral domain: the properties of Cauchy indices for rational fraction can actually be expressed using this language only, and they are connected to the properties of pseudo-reminder sequences. Here as well, the formal development essentially follows the reference textbook by Basu, Pollack and Roy [23], based on the computation of numbers coined Tarski queries. A complete account of this formal proof, as well as a description of the formal theory associated with the structure of real closed fields, is available in the corresponding paper [51].

2.3 A formal proof of the Odd Order theorem

A complete description of this formalization work is largely out of the scope of the present memoir. This section thus provides some context and motivation for the formalization endeavor undertaken by Georges Gonthier in 2006, with the Mathematical Components team. Then, after a sketch of its mathematical proof, I provide few comments about the formal statement of the theorem, and about the formalization choices for the basic vocabulary of finite group theory.

2.3.1 Context and motivation

The algebraic structure of group is a fascinating piece of abstract algebra, for the deep behavior captured by this short and simple list of axioms. Famously, Klein proposed to view geometry as the study of properties that remain invariant under the action of a given group of transformations, in his seminal Erlangen research program. Transposing the power of this nature of abstraction in a library of formalized mathematics is a far-reaching challenge.

The Odd Order theorem is a result of finite group theory, the theory of groups with a finite carrier. *Simple groups* are the most elementary, atomic instances of groups and any other finite group can be built from the finite simple ones –like molecules from these atoms. The classification of finite simple groups describes precisely the shape that a finite simple group can take, for each possible cardinal, and the Odd Order theorem, due to Feit and Thompson [72] is a corner stone of this edifice:

Theorem 2.1 *Every finite group of odd order is solvable.*

As a direct consequence, the Odd Order theorem actually provides a complete description of the structure of simple groups with an odd number of elements: they are necessarily cyclic, i.e., generated by a single element. The simplicity of the statement of this result strikingly contrasts with the sophistication of its proof,

which calls for a combination of arguments of local analysis and of character theory of finite groups. The original proof [72] was published as a 250 page monolithic paper, a record at the time. Bender, Glauberman and Peterfalvi later provided a second-generation proof [24, 142]. R. Solomon describes its impact by saying that: "This short sentence and its long proof were a moment in the evolution of finite group theory analogous to the emergence of fish onto dry land. Nothing like it had happened before; nothing quite like it has happened since" [158]. The validity of the Odd Order Theorem itself has never been really called into question, but the story of the classification of finite simple groups has been much more uneven and controversial, with notorious skeptics, including J. P. Serre [46]. As of today, the confidence of mathematicians in this complex edifice still rests more upon the reputation of the rare experts who are able to understand this composite proof *in extenso* than on a genuine assimilation by the community.

In 2006, Georges Gonthier initiated a collective research effort for formalizing a proof proof of theorem 2.1, and this endeavor would come to success 6 years later. The motivation behind this project was not to track petty errors and holes in this proof. The relevance of this proof as a case study for the formalization of mathematics, besides the significance of the result, is the broad spectrum of algebraic theories in its prerequisite, like graduate-level linear and multilinear algebra, Galois theory, representation theory and character theory of finite groups, constructions of algebraic closures, etc. The challenge was hence to represent formally all the mathematical objects that play a role in this proof: starting from the definition of natural numbers, and covering the 250 pages of the proof, plus all the pre-requisite in-between.

2.3.2 Overview

This section explains the vocabulary involved in Theorem 2.1 and give a bird-eye view of the proof. Although we have already discussed a few concepts related to finite group theory in Section 2.1.3, we recall here from scratch a few standard notations.

A *group* G consists of a set, usually also named G, together with an associative binary law $*$, usually denoted by juxtaposition, and an identity element 1, such that each element g of G has an inverse g^{-1}, satisfying $gg^{-1} = g^{-1}g = 1$. When there is no ambiguity, we identify an element g of a group with the corresponding singleton set $\{g\}$. In particular the trivial group $\{1\}$ is denoted by 1. The cardinality of G is called the order of the group. Examples of finite groups include the cyclic group $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n under addition, with identity 0; the set S_n of permutations of $\{0, \ldots, n-1\}$, under composition; and the set of isometries of a regular n-sided polygon. These examples have order n, $n!$, and $2n$, respectively.

The cartesian product $G_1 \times G_2$ of two groups G_1 and G_2 is canonically a group with law $(a_1, a_2) * (b_1, b_2) := (a_1 b_1, a_2 b_2)$; the group $G_1 \times G_2$ is called the direct product of G_1 and G_2. The law of an abelian group is commutative; in a non-abelian group G, we only have $ab = ba^b = ba[a, b]$, where $a^b := b^{-1}ab$ is the b-conjugate of a, and $[a, b] := a^{-1}b^{-1}ab$ is the commutator of a and b. Product and conjugation extend to subsets A, B of a group G, with $AB := \{ab \mid a \in A, b \in B\}$ and $Ab := \{ab \mid a \in A\}$. A subset A of G is B-invariant when $Ab = A$ for all b in B; in that case we have $AB = BA$. One says that H is a subgroup of a group G, and writes $H < G$, when H is a subset of G containing 1 that is closed under product and inverses – thus itself a group. For finite H , $H < G$ is equivalent to $1 \cup H^2 \cup H \cup G$. The set of subgroups of G is closed under intersection, conjugation, and commutative product (such as product with an invariant subgroup). If G is finite and $H < G$,then the order of H necessarily divides the order of G. It is not generally the case that for each divisor of the order of G there exists a subgroup of G of this order, but if G is a group of order n and p is a prime number dividing n with multiplicity k, then there exists a subgroup of G having order p^k, called a Sylow p-subgroup of G.The notion of a normal subgroup is fundamental to group theory:

Definition 2.2 *(Normal subgroup). H is a normal subgroup of a group G,denoted $H \triangleleft G$, when H is a G-invariant subgroup of G.*

If $H \triangleleft G$, the set $\{Hg \mid g \in G\}$ of H-cosets is a group, as $(Hg_1)(Hg_2) = H(g_1 g_2)$. This group, denoted G/H, is called the quotient group of G and H because it identifies elements of G that differ by an element of H. If G_1 and G_2 are groups, G_1 and G_2 are both normal in the group $G1 \times G2$. Every finite abelian group is isomorphic to a direct product of cyclic groups $\mathbb{Z}/p_i^k Z$, where the p_i are prime numbers. The far more complex structure of non-abelian groups can be apprehended using an analogue of the decomposition of a natural number by repeated division:

Definition 2.3 *(Normal series, factors). A normal series for a group G is a sequence $1 = G_0 \triangleleft G_1 \cdots \triangleleft G_n = G$, and the successive quotients $(G_{k+1}/G_k)_{0 \leq k < n}$ are called the factors of the series.*

A group G is simple when its only proper normal subgroup is the trivial group 1, i.e., if its only proper normal series is $1 \triangleleft G$. A normal series whose factors are all simple groups is called a composition series. The Jordan-Hölder theorem states that the (simple) factors of a composition series play a role analogous to the prime factors of a number: two composition series of the same group have the same factors up to permutation and isomorphism. Unlike natural numbers, however, non-isomorphic groups may have composition series with isomorphic factors. The class of solvable groups is characterized by the elementary structure of their factors:

Definition 2.4 *(Solvable group). A group G is solvable if it has a normal series whose factors are all abelian.*

Subgroups and factors of solvable groups are solvable, so by the structure theorem for abelian groups, a finite group is solvable if and only if all the factors of its composition series are cyclic of prime order. As a consequence of Theorem 2.1, a finite simple group of odd order is both simple and solvable, and thus cyclic.

2.3.3 Proof sketch

This section is an excerpt from our article [81]. It uses more advanced concepts in finite group theory, and can be safely skipped.

The proof of Theorem 2.1 proceeds by induction, showing that no minimal counterexample G exists. At the outset G is only known to be simple, non-abelian of odd order, but all proper subgroups of G should be solvable. The first half of the proof exploits these meager facts to derive a detailed description of the maximal proper subgroups of G, reducing the general structure of G to five cases. The second half of the proof uses character norm inequalities to rule out four of these, and extract some algebraic identities in a finite field from the last one. Galois theory is then used to refute these, completing the proof.

The study of the (solvable) subgroups of G exploits their decomposition into prime factors, reconstructing the structure of a maximal subgroup M from that of its p-factors for individual primes p. An A-invariant subgroup H of M has a normal series with A-invariant elementary abelian factors, that is, direct products of prime cycles. Identifying each one with a vector space over a finite field $\mathbb{F}_p$ makes it possible to analyze the action of A on H via the representations mapping A to a group of matrices over $\mathbb{F}_p$, and use linear algebra techniques such as eigenspace decomposition. Indeed, the proof starts by showing that 2×2 representations are abelian, then that no representation of A has a quadratic minimal polynomial (this replaces the use of the Hall-Higman theorem in [72]). This p-stability is combined with Glauberman's ZJ^* factorization to establish a Uniqueness theorem (Chapter II of [24]): any subgroup of rank 3 (containing an elementary abelian subgroup of order p^3) lies in a unique maximal subgroup of G.

Combining the Uniqueness theorem with results of Blackburn on odd groups of rank 2 yields that any maximal subgroup M of G is a semi-direct product $M_\sigma \rtimes E$ with $M_\sigma \triangleleft M$ and M_σ, E of coprime order. Furthermore, very few elements of M_σ and E commute – M is similar to a Frobenius group. Further analysis reveals that most M are of type I: M is very nearly a Frobenius group, with M_σ equal to the direct product M_F of the normal Sylow subgroups of M. However some M can be of type P, with $M = M_F U W_1$, where W_1 is cyclic, $U W_1$ is a Frobenius group,

and all w_1 in W_1 commute precisely with the same cyclic group $W_2 < M_F$ (W_1 acts in a prime manner on M_F). Type P is subdivided into types V, II, III or IV, according to whether U is trivial, included in a different maximal group, abelian, or non-abelian, respectively. If any, there are exactly two type P groups up to conjugation, with W_1 and W_2 interchanged; at least one has type II, and over half of the elements of G lie in conjugates of $W = W_1W_2$.

The second part of the proof [142] uses characters. The character of a complex representation $\rho : H \mapsto GL(n, \mathbb{C})$ is the function mapping each $h \in H$ to the trace of $\rho(h)$. In general, a character is not a group homomorphism, but it is a class function, constant on conjugacy classes of H. Convolution over H makes the set of class functions on a group H into a Hermitian space, for which the set irr H of irreducible characters of H forms an orthonormal basis. Characters of H have natural integer coordinates in irr H, hence an integral norm.

Local analysis provides us both with a precise description of the characters of a maximal subgroup M, and an isometry mapping certain virtual characters of M (differences of characters) to virtual characters of G. This Dade isometry is only defined on functions that vanish on 1, so in order to extract usable information on G one needs coherence theorems extending it to a set of proper characters. The first, due to Sibley, covers Frobenius and type V maximal sub-groups, and the second type II–IV subgroups. For any $x \in$ irr G, coherence for a set (M_i) of non-conjugate maximal sub-groups implies a numerical inequality bounding the sum of the (Hermitian) norms of the inverse images of the restrictions of x to the support of the image of the Dade isometries for the M_i, and the (complex) norms of the values of x elsewhere. For types III–V this bound yields a non-coherence theorem, which successively eliminates types V and IV; this implies that type I groups are actually Frobenius, and then the coherence bound forces type P groups to exist.

More inequalities then force the M_F, U, and W_1 subgroups of the type P groups to be isomorphic to the additive, unitary multiplicative, and Galois groups of a finite field $\mathbb{F}_{p^q}$ of order p^q, then rule out type III, and imply that U is W_2^y-invariant for some $y \in H_F$, where H is the other type II group such that $W_1 < H_F$. Intricate calculations show that this implies that if $a \in \mathbb{F}_{p^q}$ and $2 - a$ both have Galois norm 1, then so does $\tau(a) := 2 - 1/a$, and hence $\tau(a)...\tau^k(a) = (1 - 1/a)k + 1$; for $a \neq 1$ the Galois norm of $(1 - 1/a)x + 1$ yields a polynomial of degree q which has $0, \ldots, p - 1$ as roots, whence $q \leq p$ and hence $q = p$ by symmetry, so the orders of M_F and $E > W_1$ are not coprime, a contradiction.

2.3.4 Formalization of the statement

The source code devoted to the verification of the two volumes, respectively by Bender and Glauberman [24] and Peterfalvi [142], represents 40633 lines of code, comments included (6% of the code, on average). Each chapter of each of the two aforementioned books corresponds to one file in the archive: the material in chapter 1 of Bender and Glauberman's book is formalized in file `BGsection1.v` (and so on for the 16 chapters), the one of chapter 1 of Peterfalvi's book is formalized in file `PFsection1.v` (and so on for the 14 chapters). Comments keep track of the correspondence between lemma numbering in the paper proof and formal statements. The complete source code is available online and maintained to keep up with the evolution of Coq and of the Mathematical Components library.

The final statement of the theorem, to be found in file PFsection14 is the following:

```
Theorem Feit_Thompson (gT : finGroupType) (G : {group gT}) :
  odd #|G| -> solvable G.
```

The statement involves the `fingroupType` group domain type mentioned in Section 2.1.3, and quantifies over (finite) groups with an arbitrary group domain type. For the skeptical, a stripped version of the formal statement provides a wording of the same result which does not make use of any advanced elaboration feature of the Coq proof assistant, like notations, implicit argument, etc.

```
Theorem stripped_Odd_Order T mul one inv
  (G : T -> Type) (n : natural) :
  group_axioms T mul one inv -> group T mul one inv G ->
  finite_of_order T G n -> odd n ->
  solvable_group T mul one inv G.
```

In fact, this statement does not even use the *prelude* of Coq, i.e. the set of libraries silently and automatically loaded by the proof assistant when a session is open. Instead, the file where it is stated also contains copies of the inductive types involved in the definitions, which would usually be retrieved from this prelude, like equality or Peano natural numbers. In this version, a (finite) group G is formalized as the characteristic function of a collection of elements in a pointed type T, equipped with a binary operation (for the product) and a unary one (for the inverse). The `group_axioms` predicate ensures that the product is associating and that the inverse is a left and right inverse o the latter. The `group` predicate ensures that G contains the unit and is closed under inverse and product. Finally the `finite_of_order` predicate states that G is finite of cardinal n. Solvability is defined inductively, from the definition of an abelian factor. The definition of this stripped version, as well as its proof from the one of the previous version, takes about 210 lines of code in total.

2.4 Deep-embedding

This section focuses on the usage of deep-embedding, also called reification techniques, in the Mathematical Components libraries, in some parts rather specific to the proof of the Odd Order theorem. Maybe more than in any other part of this memoir, my contribution to the work presented in this section is more in the description and documentation of the existing, than in the actual formalization work. However, I chose to mention this topic because it provides a nice application to the material presented in Section 2.2, but also, incidentally, because it echoes to my first publication in interactive theorem proving [87], which used reification techniques for large scale, formal-proof producing, automated proving.

2.4.1 Deep-embedding and constructive proofs

The variant of proof of the Odd Order Theorem formalized in this work relies on nothing but the axioms and rules of the Calculus of Inductive Constructions implemented by Coq[3]. In particular, this variant, as well as all the theories it depends on, is constructive. Apart from the fact that reasoning by contradiction is not available on arbitrary statements, this perspective is however mostly visible in the lowest layers of the libraries. Some of these layers would just vanish in a non-constructive context. For instance, in the presence of a global and sufficiently strong choice axiom, the structure for types equipped with a choice operator present in the hierarchy would be vacuous, for it would endow any type.

In this proof, each usage of a choice principle can be reduced to choice over a countable (possibly finite) type, a variant of choice axiom which is provable in the dependent type theory implemented by Coq.

An obvious deviation from the standard literature induced by a constructive viewpoint is the increased precision needed for describing real and complex numbers. Number fields, and more generally sub-fields of $\mathbb{C}$, come into play in the proof when representation theory is used to study the characters of the postulated minimal counter example. In fact, in this proof, any occurrence of $\mathbb{C}$ can be harmlessly, and usefully, replaced by the algebraic closure $\tilde{\mathbb{Q}}$ of $\mathbb{Q}$. Equality is decidable for points in $\tilde{\mathbb{Q}}$, and in fact its full ring first-order theory is decidable, as it is an algebraically closed field. This motivated a formal construction of $\tilde{\mathbb{Q}}$, due to Georges Gonthier. The construction is performed in two stages. First comes a construction of an algebraically closed field equipped with an order 2 field automorphism:

```
Theorem Fundamental_Theorem_of_Algebraics :
```

[3] Actually, it uses a strict subset of the features available in Coq and does not rely, e.g., on co-inductive types nor on universe polymorphism.

```
{L : closedFieldType &
{conj : {rmorphism L -> L}|involutive conj & ~ conj =1 id}}.
```

This statement provides an actual witness of closed field `L` and a ring endomorphism `conj` which is is involutive but not the identity. Then, $\tilde{\mathbb{Q}}$ is constructed as a partially ordered sub-field of the latter, in which the order 2 field automorphism plays the expected conjugation role:

```
Variables (L : closedFieldType) (conj : {rmorphism L -> L}).
Lemma ComplexNumMixin :
involutive conj -> ~ conj =1 id ->
{numL : numMixin L |
  ∀x : NumDomainType L numL, `|x| ^+ 2 = x * conj x}.
```

This statement is parameterized by a closed field `L` and an order 2 automorphism, and provides a witness of partially ordered subfield with an absolute value, which behaves as expected regarding conjugation.

Note that this closure construction applies in fact to any field with a countable number of inhabitants, and a decidable equality. Recently, Paulo Emílio de Vilhena and Lawrence C. Paulson have formalized the general construction of the algebraic closure in the classical setting of the Isabelle/HOL proof assistant [60].

Once constructed, by applying the previous construction to the type of rational numbers, the field $\tilde{\mathbb{Q}}$ benefits from the formalized quantifier elimination procedure for algebraically closed fields, described in Section 2.2, and thus from the proof of decidability of first order statements in the language of rings with constants in $\tilde{\mathbb{Q}}$. As a consequence, it is possible to reason constructively by case analysis on the validity or a properties which can be expressed in this language.

A concrete example where this is useful is the definition of socles, for representations of finite groups: this definition involves testing submodules for simplicity with respect to a given representation r. By definition, an r-submodule is a module invariant under r, and it is said to be simple if it is minimal with respect to this stability property. Simplicity can be expressed as a first order statement, and thus formalized as a boolean term if the corresponding first theory is decidable [4]. This is the case for a modular representation, because the language is the first order theory of a finite field, but also in our setting for a complex representation, since $\tilde{\mathbb{Q}}$ can be taken as the base field.

Below is a brief illustration of this formalization pattern with an excerpt of the library `mxrepresentation`. Consider a finite dimensional vector space U over a field F, of dimension smaller than n, and a representation r_G of a finite group G into

[4] We only consider modules of finite type here.

square matrices of size $n \times n$ and coefficients in F. The definition `mxnonsimple`, in sort `Prop`, states that U has a non-trivial strict r_G-submodule. Now the term `(mxnonsimple_form rG (mx_term U))` is a reified, quantifier free, version of the same statement, as deep-embedded first-order formula. When the field F moreover has a decidable first-order theory, the satisfiability of the latter formula is a boolean value, and this is the definition `mxnonsimple_sat`. Finally, lemma `mxnonsimpleP` establishes the equivalence of the two versions for a non-trivial module U. The term `mxnonsimple_sat` is thus a "decision procedure" for testing the presence of a simple r_G-submodule in a non-trivial module U.

```
Variables (F : decFieldType) (n : nat)  (U : 'M_n(F)).
Variables (G : {group gT}) (rG : mx_representation F G n).

Definition mxnonsimple : Prop := ∃V : 'M_n,
[&& mxmodule rG V, (V <= U)%MS, V != 0 & \rank V < \rank U].

Definition mxnonsimple_sat : bool :=
GRing.sat (@row_env _ (n * n) [::])
          (mxnonsimple_form rG (mx_term U)).

Lemma mxnonsimpleP :
  U != 0 -> reflect mxnonsimple mxnonsimple_sat.
```

This wording allows to prove that any non-trivial r_G-module U has a simple r_G-submodule by (induction on the rank of U and) case analysis on the simplicity of U, as one would expect.

In other cases, first-order decidability fails, notably for the first order theorem of the rationals and for that of number fields. As a result, we elected not to rely on this interface for some basic results in the theory of group modules, that cannot be proved constructively. Instead, we proved their double negation, expressed using a `classically` monadic operator [80, 138]:

```
Definition classically (P :Prop) : Prop :=
  ∀b : bool, (P -> b = true) -> b = true.
```

The statement `(classically P)` is logically equivalent to `(~~ P)`, but this formulation is more handy in practice, because when using a hypothesis of the form `(classically P)` in the proof of a statement expressed as a boolean (i.e., on which excluded middle holds), one can constructively assume that `P` itself holds. Here is an example of such a classical result, which asserts the existence of simple submodule V (in fact, a vector space) for any non trivial subspace U of K^n, with K an arbitrary field:

```
Variables (K : fieldType) (n : nat).
Lemma mxsimple_exists (m : nat) (U : 'M_(m, n)) :
```

```
mxmodule U -> U != 0 ->
classically (exists2 V, mxsimple V & V <= U)%MS.
```

2.4.2 Deep embedding and group presentations

We have seen how to make use of a deep embedding of first-order logic to justify the use of excluded middle on the evaluation of a reified statement. This nature of quotation finds another application in the formalization of groups defined by presentations. Presentations are mainly useful for the study of a class of finite groups coined *extremal* p-groups, which are non-abelian with a cyclic maximal subgroup, in which the order of every element is a power of the prime p. A structure theorem provides a classification of extremal p-groups into four families (modular, dihedral, generalized quaternion and semi-dihedral). In particular, for each family, the theorem provides a *presentation* describing the groups in this family.

Presentations of groups appear in two different guise. The first one considers presentations as constructing a group by quotient of a free group by relations. Here we follow Aschbacher's Finite Group Theory [13], but for the notations. For any cardinality C, there is, up to isomorphism, a unique free group with a free generating set of cardinal C. Then, if Y is a set distinct symbols, and W a set of words on alphabet $Y \cup Y^{-1}$, then one writes $\langle Y \mid W \rangle$ for "the" group F/N, where F is the free group generated by Y and N is the normal subgroup of F generated by the subset W. The group $\langle Y \mid W \rangle$ is thus the group generated by Y, subject to the relations $w = 1$ for $w \in W$, otherwise said, the largest group generated by Y in which $w = 1$ for each $w \in W$. When a group has a presentation with a finite set of generators and relators, then it is said to be finitely presented, and we will only consider such finite presentations below. But even in this finitary setting, the construction of a group with a given presentation is not effective. In fact, whether two finitely presented groups are isomorphic or not, as well as whether or not a given presentation is that of trivial groups, is undecidable [6, 145].

However, for the purpose of stating, proving and using theorems akin to the classification of extremal p-groups, this lack of effectivity is however irrelevant, as presentations are merely used in this context to describe a property for pre-existing groups. Following the same reference by Aschbacher [13], a presentation for a group G is a set Y of generators of G, together with a set W of words on alphabet $Y \cup Y^{-1}$, such that the relation $w = 1$ is satisfied in G for each $w \in W$, and the natural homomorphism of $\langle Y \mid W \rangle$ onto G is an isomorphism. In this case, one writes:

$$G = \langle Y \mid W \rangle$$

For instance for any group G, $\langle g \in G, xy(xy)^{-1} \rangle$ is a presentation for G. Perhaps more interesting, the family of dihedral groups of order $2n$, denoted D_{2n} also enjoy a simple presentation. The group D_{2n}, is defined as the semi-direct product of a cycle $\langle r \rangle$ of order n with a cycle $\langle s \rangle$ of order 2, via the automorphism $\varphi_s(r) = r^{-1}$. The group D_{2n} can also be seen as the set of isometries leaving a regular n-sided polygon invariant, in which case r corresponds to a rotation of angle $\frac{2\pi}{n}$, and s to a reflection. It is easy to see that D_{2n} is presented as:

$$\langle x, y \mid x^n, y^2, srsr^{-1} \rangle$$

The `presentation` Coq library only formalizes this second usage, which views the statement:

$$G = \langle Y \mid W \rangle$$

as a predicate on groups, here applied to a concrete group G. The formal definition of this predicate is built from two inductive data structures, respectively for providing the language used to provide relators, and for the corresponding formulas, which are just iterated conjunctions of equalities between terms:

```
Inductive term :=
  | Cst of nat
  | Idx
  | Inv of term
  | Exp of term & nat
  | Mul of term & term
  | Conj of term & term
  | Comm of term & term.
```

Note that the language for reified terms is not at all minimal, as it rather intends to offer a convenient language for the user to describe relators. Now a presentation is given by a formula, which lists its defining relations:

```
Inductive formula :=
  Eq2 of term & term | And of formula & formula.
```

The language of formula slightly deviates from Aschbacher's definition, in that it allows to define relators as equations between terms, rather than imposing one of the terms to be the neutral element.

Given a group domain `(gt : finGroupType)`, and a list `e` of points in this domain used as a context for interpreting constants, any term `(t : term)` can be interpreted as a point in `gT` via the following straightforward evaluation function, defined by induction on the syntax of the term:

```
Variable (gT : finGroupType).
Fixpoint eval (e : seq gT) (t : term) : gT :=
```

```
  match t with
  | Cst i => nth 1 e i
  | Idx => 1
  | Inv t1 => (eval e t1)^-1
  | Exp t1 n => eval e t1 ^+ n
  | Mul t1 t2 => eval e t1 * eval e t2
  | Conj t1 t2 => eval e t1 ^ eval e t2
  | Comm t1 t2 => [~ eval e t1, eval e t2]
  end.
```

and a boolean test for the validity of a given formula in a given subset of a group domain is built from this evaluation function.

The presentation of dihedral groups D_{2q}, for $q > 1$, is formalized as:

```
Variable q : nat.
Hypothesis q_gt1 : q > 1.
Let m := q.*2. (* m := 2 * q *)

Lemma Grp_dihedral :
  'D_m \isog Grp (x : y : (x ^+ q, y ^+ 2, x ^ y = x^-1)).
Proof. ... Qed.
```

where the notation `'D_m` refers to the explicit construction by semi-direct products of $\mathbb{Z}/q\mathbb{Z}$ by $\mathbb{Z}/2\mathbb{Z}$. This sentence seemingly overloads the infix notation for isomorphism of finite groups (using a dedicated scope). But in fact, the notation refers to the predicate

```
  _ \isog Grp (x : y : (x ^+ q, y ^+ 2, x ^ y = x^-1))
```

which is built from a reified presentation formula, and from its evaluation. In fact, the statement of lemma `Grp_dihedral` unfolds to the following formula:

```
  ∀(rT : finGroupType) (H : {group rT}),
  (H \homg 'D_m) =
  (H \homg Grp (x : y : (x ^+ q, y ^+ 2, x ^ y = x^-1)))
```

where the two occurrences of the infix `\homg` notation refer to two different acceptations. The one in the left hand-side, `(G \homg 'D_m)`, refers to the existence of a homomorphism from `H` to `'D_m`. The one in the right hand-side, `(H \homg Grp (x : y : (x ^+ q, y ^+ 2, x ^`
overloads the latter, and unfolds to the following boolean expression:

```
  [∃t : rT * rT, let: (r, s) := t in
    [&& (<[r]> <*> <[s]> == H) ,
        (r ^+ q == 1), (s ^+ 2 == 1) & (r ^ s == 1)]
  ]
```

which tests the presence in the finite type `rT` of two points `r` and `s`, such that `H` is equal as a set to the subgroup generated by `r` and `s`, and such that the expected

equations hold. The presentation holds if and only if these two acceptations are equivalent, that is, if the two corresponding boolean values are equal for any H.

2.5 Conclusion and perspectives

This chapter provided a very partial and biased selection of topics related to the Mathematical Components library. The Mathematical Components eponymous book [126] provides a more in-depth description of the design patterns that are adopted in a uniform way throughout the library. However, a more thorough description of the formalized mathematical theories available in this library, easier to crawl than the existing collection of header comments, is still missing from the picture. This nature of documentation is in fact quite difficult to design for Coq libraries, and, more importantly, hard to maintain. In this respect, the documentation of the mathlib library [3] for the Lean proof assistant represents the state of the art, although the library is arguably much younger. The Archive of Formal Proofs (AFP) of the Isabelle/HOL proof assistant provides another successful model of long-lasting collaborative and curated development, which is "organized in the way of a scientific journal"[5] and provides accompanying documentation and citation facilities for each of its entries.

One of the delicate issues to be addressed when designing such a large corpus of formalized mathematics is the one of abstraction and modularity. In particular, it is crucial to devise a working hierarchy of structures, for abstract algebra, which can be populated with as many instances as needed. The tools of the trade are tied to the underlying logical formalism and we have focused here on the case of dependent type theory, although there is a related literature for systems based on other foundations, notably HOL [17, 89, 101]. For the proof of the Odd Order theorem, using packed classes [78] in a systematic proved a successful approach: incidentally, Galois theory represents an interesting test case for a hierarchy as reasoning with field extensions superposes ingredients from commutative algebra and from linear algebra on a same carrier type. Less bundled approaches, like the one advocated by the MathClasses library [161] can trigger difficult efficiency and control issues. Yet a serious limitation of the current hierarchy implemented in the Mathematical Components library is the fact that it is very difficult to modify and to extend: the size of the required boilerplate code corresponding to an additional node in the graph of structures grows rapidly, and error messages are quite difficult to interpret in case of mistake. Recent contributions for debugging [154] hierarchies based on packed classes, and for generating [52] the corresponding code to a newly added structure should significantly improve the situation. But more

[5] https://www.isa-afp.org/

satisfactory implementations might only be possible from a better understanding of the formalization of categories in dependent type theory.

As such, the Mathematical Components library has nonetheless been used by users outside the development team as a backbone library for a wide range of applications in real algebraic geometry [63], robotics [150], verified probabilistic data-structures [84], combinatorics[6], graph theory [64], discrete geometry [9], etc. Among the future directions of work that would help making this range of applications even wider are the improvement of automation and the implementation of additional consolidated formalized volumes for undergraduate real and complex analysis.

[6] https://github.com/hivert/Coq-Combi

Chapter 3

Symbolic computations

3.1 Proofs by symbolic computations

According to the reference textbook by Bostan et al. [35], symbolic computation, or computer algebra, "*studies* exact *mathematical objects, from a* computer science *viewpoint*. In this context, *exact* means that the theories of interest are mainly equational, i.e., express identities rather than estimations. And a *computer science viewpoint* focuses on effective methods, on the study of their complexity, and on the design of optimal algorithms."[1] From the late 60s on, symbolic computation gained traction, and eventually emerged as a scientific area of its own, getting fame with the Risch algorithm [148] for indefinite integration. It gradually provided efficient computer implementations and got attention in experimental mathematics. Beside commutative algebra, differential and recurrence equations have remained a central research topic of computer algebra over the years.

In order to operate on sequences (or more generally on functions), computer algebra substitutes implicit representations for explicit representations in terms of named sequences (factorial, binomial, etc): Bruno Salvy's survey [156] provides a panorama of this fertile viewpoint. In this vein, ∂-finite functions (and sequences) provide an emblematic example of algebraic objects which enjoy nice algorithmic properties, while encoding interesting properties for a rather large class of functions. Roughly speaking, ∂-finite sequences (resp. functions), are solution of a linear recurrence (resp. differential) system, with polynomial coefficients. Notably, the finiteness property of their definition makes algorithmic most operations under which the class of ∂-finite sequences is stable. See for instance Manuel Kauers' survey for an introduction to the topic [111].

[1] Original text in French, translation is mine.

Symbolic computations can go as far as providing proofs, and sometimes even provide the sole known proofs of some results: a notable example being the proof by computer algebra of the q-TSPP conjecture, proved by Manuel Kauers, Christoph Koutschan and Doron Zeilberger [113]. The latter author is indeed a notorious advocate of computer-generated proofs. His book on the topic [143], co-authored with Marko Petkovsek and Herbert Wilf, contains elegant recipes to establish combinatorial identities "by computer algebra", in particular those involving hypergeometric sequences, which are a particular case of ∂-finite sequences.

This chapter illustrates an application of symbolic computation on ∂-finite sequences to a famous problem in number theory. The computations involved in this computer proofs have the pleasant feature to be easy to check *a posteriori*, either by a normalization procedure, possibly with the help of certificates output by the algorithms at stake. This feature was part of the motivation for starting a formal verification of this nature of calculation: it was an appealing perspective to have the expensive exploration of search space performed by efficient computer-algebra programs, and to be able to control the amount of formally verified computation needed for a completely verified proof. As discussed in the corresponding publication [47], the formal proof is a standalone collection of Coq source files, but some of them have been generated by the Maple/Algolib computer algebra system, and manually annotated with extra information (provisos for recurrence relations) needed for a complete proof.

3.2 A formal proof of Apéry's theorem: context

In 1978, Roger Apéry proved that $\zeta(3) := \sum_{i=1}^{\infty} \frac{1}{i^3}$, now known as the *Apéry constant*, is irrational. This result was the first dent in the problem of the irrationality of the evaluation of the Riemann ζ function at *odd* positive integers. As of today, this problem remains a long-standing challenge of number theory. Rivoal [149] and Zudilin [184] showed that at least one of the numbers $\zeta(5)$,$\zeta(7)$,$\zeta(9)$, and $\zeta(11)$ must be irrational. Fischler, Sprang and Zudilin have proved that many odd zeta values are irrational [76]. But today $\zeta(3)$ is the only one *known* to be irrational.

Van der Poorten [174] reports that Apéry's announcement of this result was at first met with wide skepticism. His obscure presentation featured "a sequence of unlikely assertions" without proofs, not the least of which was an enigmatic recurrence (Lemma 3.4) satisfied by two sequences a and b. It took two months of collaboration between Cohen, Lenstra, and Van der Poorten, with the help of Zagier, to obtain a thorough proof of Apéry's theorem:

Theorem 3.1 (Apéry, 1978) *The constant $\zeta(3)$ is irrational.*

There exist today several other proofs of Apéry's theorem, notably a concise and elegant proof by Fritz Beukers [29], published shortly after Apéry's announcement. This gallery includes several *computer proofs*, which use runs of computer algebra algorithms to synthesize part of the argument. This approach was initiated by Zeilberger [182], illustrating the potential of his *creative telescoping algorithms* [48].

Today, a complete formal proof of this theorem is available, formalized using the Coq proof assistant and the Mathematical Components libraries [126]. This formalization follows the structure of Apéry's original proof. However, symbolic computations replace the manual verification of recurrence relations, via an automatic discovery of these equations. More precisely, Maple packages perform calculations outside the proof assistant, and the resulting claims are verified *a posteriori*, using Coq. A complete (and constructive) formal proof of Theorem 3.1 follows from combining the latter verified claims with some additional formal developments. The formal statement of this theorem, in Gallina, is:

```
Theorem zeta_3_irrational : ~ (exist r : rat, z3 == r%:CR).
```

where `z3` is the formal definition of the real number $\sum_{i=1}^{\infty} \frac{1}{i^3}$.

3.3 Overview of the proof

According to Stéphane Fischler' survey [75], all known proofs of Apéry's theorem share a common structure. They rely on the asymptotic behavior of the sequence ℓ_n, the least common multiple of integers between 1 and n, and they proceed by exhibiting two sequences of rational numbers a_n and b_n, which have the following properties:

1. For a sufficiently large n:
$$a_n \in \mathbb{Z} \quad \text{and } 2\ell_n^3 b_n \in \mathbb{Z};$$
2. The sequence $\delta_n = a_n\zeta(3) - b_n$ is such that:
$$\limsup_{n\to\infty} |2\delta_n|^{\frac{1}{n}} \leq (\sqrt{2}-1)^4;$$
3. For an infinite number of values n, $\delta_n \neq 0$.

Altogether, these properties entail the irrationality of $\zeta(3)$. Indeed, if we suppose that there exists $p, q \in \mathbb{Z}$ such that $\zeta(3) = \frac{p}{q}$, then $2q\ell_n^3\delta_n$ is an integer when n is large enough. One variant of the Prime Number theorem states that $\ell_n = e^{n\,(1+o(1))}$ and since $(\sqrt{2}-1)^4 e^3 < 1$, the sequence $2q\ell_n^3\delta_n$ has a zero limit,

which contradicts the third property. Actually, the Prime Number theorem can be replaced by a weaker estimation of the asymptotic behavior of ℓ_n, that can be obtained by more elementary means.

Lemma 3.2 *Let ℓ_n be the least common multiple of integers $1 \dots n$, then*

$$\ell_n = O(3^n).$$

Since we still have $(\sqrt{2}-1)^4 3^3 < 1$, this observation [91, 73] is enough to conclude.

In our formal proof, we consider the pair of sequences proposed by Apéry in his proof [12, 174]:

$$a_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2, \qquad b_n = a_n z_n + \sum_{k=1}^{n} \sum_{m=1}^{k} \frac{(-1)^{m+1} \binom{n}{k}^2 \binom{n+k}{k}^2}{2m^3 \binom{n}{m} \binom{n+m}{m}} \tag{3.1}$$

where z_n denotes $\sum_{m=1}^{n} \frac{1}{m^3}$, a sequence obviously converging to ζ_3.

By definition, a_n is a positive integer for any $n \in \mathbb{N}$. The integrality of $2\ell_n^3 b_n$ is not as straightforward, but rather easy to see as well: each summand in the double sum defining b_n has a denominator that divides $2\ell_n^3$. Indeed, after a suitable re-organization in the expression of the summand, using standard properties of binomial coefficients, this follows easily from the following slightly less standard property:

Lemma 3.3 *For any integers i, j, n such that $1 \leq j \leq i \leq n$, $j\binom{i}{j}$ divides ℓ_n.*

Proof This can be obtained as a direct corollary of a classical formula for the p-valuation of the factorial $n!$, for p a prime number and $n \in \mathbb{N}$:

$$v_p(n!) = \sum_{i=1}^{\lfloor \log_p n \rfloor} \left\lfloor \frac{n}{p^i} \right\rfloor.$$

Indeed, observe that for any prime p, the p-valuation of $j\binom{i}{j}$ is smaller than the one of ℓ_n. □

The rest of the proof is a study of the sequence $\delta_n = a_n \zeta(3) - b_n$. It is easy to see that δ_n tends to zero, from the formulas defining the sequences a and b, but we also need to prove that it does so fast enough to compensate for ℓ_n^3, while being positive. In his original proof, Apéry derived the latter facts, positivity and limit of δ_n, by combining the definitions of the sequences a and b with the study of the mysterious recurrence relation (3.2). Indeed, he made the surprising claim that Lemma 3.4 holds:

Lemma 3.4 *For $n \geq 0$, the sequences $(a_n)_{n\in\mathbb{N}}$ and $(b_n)_{n\in\mathbb{N}}$ satisfy the* same *second-order recurrence:*

$$(n+2)^3 y_{n+2} - (17n^2+51n+39)(2n+3)y_{n+1} + (n+1)^3 y_n = 0. \tag{3.2}$$

Equation 3.2 is a typical example of a linear recurrence equation with polynomial coefficients and standard techniques [155, 174] can be used to study the asymptotic behavior of its solutions. Using this recurrence and the initial conditions satisfied by a and b, one can thus obtain the two last properties of Fischler's criterion, and conclude with the irrationality of $\zeta(3)$. The formal proof relies on a simplified version of this asymptotic study, essentially a variant of the presentation by van der Poorten [174].

Note that using Equation 3.2 alone, even with sufficiently many initial conditions, it would not be easy to obtain the first property of our criterion, about the integrality of a_n and b_n for a large enough n. In fact, it would also be difficult to prove that the sequence δ tends to zero: we would only know that it has a finite limit, and how fast the convergence is. By contrast, it is fairly easy to obtain these facts from the explicit Formulas 3.1.

The proof of Lemma 3.4 was by far the most difficult part in Apéry's original exposition. In his report [174], van der Poorten describes how he, with other colleagues, devoted significant efforts to this verification, after having attended the talk in which Apéry exposed his result for the first time. But in the end, the proof of Lemma 3.4 actually amounts to a routine calculation using the two auxiliary sequences $U_{n,k}$ and $V_{n,k}$, themselves defined in terms of $\lambda_{n,k} = \binom{n}{k}^2\binom{n+k}{k}^2$ (with $\lambda_{n,k} = 0$ if $k < 0$ or $k > n$):

$$\begin{aligned} U_{n,k} &= 4(2n+1)(k(2k+1) - (2n+1)^2)\lambda_{n,k}, \\ V_{n,k} &= U_{n,k}\left(\sum_{m=1}^{n}\frac{1}{m^3} + \sum_{m=1}^{k}\frac{(-1)^{m-1}}{2m^3\binom{n}{m}\binom{n+m}{m}}\right) \\ &\quad + \frac{5(2n+1)k(-1)^{k-1}}{n(n+1)}\binom{n}{k}\binom{n+k}{k} \end{aligned}$$

The key idea is to compute telescoping sums for U and V. For instance, we have:

$$U_{n,k} - U_{n,k-1} = (n+1)^3\lambda_{n+1,k} - (34n^3+51n^2+27n+5)\lambda_{n,k} + n^3\lambda_{n-1,k} \tag{3.3}$$

Summing Equation 3.3 on k shows that the sequence a satisfies the recurrence relation of Lemma 3.4. A similar calculation proves the analogue for b, using telescoping sums of the sequence V.

Not only is the statement of Formula 3.2 difficult to discover: even when this recurrence is given, finding the suitable auxiliary sequences U and V by hand is a difficult task. Moreover, there is no other known way of proving Lemma 3.4 than by exhibiting this nature of certificates.

Fortunately, the sequences a and b belong in fact to the class of objects discussed in Section 3.1 and are thus amenable to a proof by symbolic computation. Basing on the the Maple package Mgfun (distributed as part of the Algolib [8] library), Salvy wrote a Maple worksheet [155] that follows Apéry's original method but interlaces Maple calculations with human-written parts. In particular, this worksheet illustrates how parts of this proof, including the discovery of Apéry's mysterious recurrence, can be performed by symbolic computations. The formal proof of Lemma 3.4 follows an approach similar to the one of Salvy. It is based on calculations performed using the Algolib [8] library, and formally verified *a posteriori* using the Coq proof assistant. However, it turned out that the recurrences generated by the computer algebra program were not sufficient to provide complete proof of the desired formulas: the algebraic data produced by Maple, which represent recurrence operators, have to be annotated *a posteriori* with provisos excluding some values from the recurrence relations. Moreover, as of today these provisos are difficult to produce algorithmically, and are thus input manually, after a human inspection (see the corresponding paper for more details [47]).

3.4 Numbers and Types

The formal verification is conducted with the Coq proof assistant, and does not feature any unproved assumption: it only relies on the axioms of the logic underlying Gallina. In particular, mathematical objects are described using dependent type theory, and the proof is entirely constructive. We take again here the formal statement of Apéry's theorem given at the end of section 3.2:

```
Theorem zeta_3_irrational : ~ (exist r : rat, z3 == r%:CR).
```

In this statement, `rat` is a type representing rational numbers, as pairs of co-prime integers. Type `CR` models real numbers, as the total setoid [31, 21] of Cauchy sequences with an explicit effective modulus of convergence, equating by relation `(_ == _)` the sequences with the same limit. For now on, *real numbers* refers to this definition. The postfix notation `_%:CR` refers to an embedding of rational numbers into real numbers. The constant `z3` is the real number associated with the Cauchy sequence $(\sum_1^n \frac{1}{n^3})_{n\in\mathbb{N}}$. This theorem thus states that the real number `z3` cannot be equivalent, as a Cauchy sequence, to any rational real number.

This formal statement gives an indication of one of thorny issues of formalizing mathematics in type theory: a set-theoretic flavor of the same sentence would simply be:

$$\zeta(3) \notin \mathbb{Q}$$

Yet, when formalizing this statement in type theory, one has to attach a *type* for each of the numbers it mentions. Thus, in the type-theoretic version, the constant $\zeta(3)$ is represented as a term with type the one of real numbers. As no subtyping can play the role of the inclusion of rationals into real numbers, the reals that are rational numbers are represented effectively, as the image of a function from the (type of) rationals to real numbers: this function is what the postfix notation `_%:CR` denotes. The formal libraries for this proof make use of the following sets of numbers:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$$

each of which is represented by a different type. Each inclusion symbol in the chain is associated with an explicit embedding function: $\mathbb{N}$ is represented using the classic inductive data type for Peano, unary representation, and $\mathbb{Z}$ as a two-constructor inductive, for two copies of the latter: one for non-negative integers and one for the negative ones. The first constructor of this type provides the embedding $\mathbb{N} \subset \mathbb{Z}$. Then, a generic construction embeds a copy of the (initial) ring $\mathbb{Z}$ into any instance of ring, and in particular into the type $\mathbb{Q}$ of rational numbers. Finally, the type $\mathbb{R}$ of real numbers embeds a copy of $\mathbb{Q}$, the (equivalence classes of) constant sequences with a rational value. Note that in fact, in a small number of statements related to the estimation of asymptotic behaviors, like in the proof of Lemma 3.2, we also use a type of complex algebraic numbers. An explicit cast can turn a rational number into a complex algebraic number, but of course no such cast exist for arbitrary real numbers.

While typing casts and typing constraints are key to implementing powerful generic notations, these explicit embedding can become tricky to work with. For instance, generic casts like the embedding of $\mathbb{Z}$ in any instance of ring cannot be declared as coercions, if only because they do not meet the syntactic requirements imposed by Coq. They should thus be inserted by hand, and have to be explicitly displayed in formulas: the postfix notation `_%:CR` is thus an attempt at a lightweight display of an information most often left implicit on paper. On the other hand, the embedding $\mathbb{N} \subset \mathbb{Z}$ can and is declared as a coercion, automatically and silently inserted by Coq. Unfortunately, this does not imply an automated management of morphism identities: in the worst cases, two non-convertible expressions can be displayed exactly the same, which can significantly hamper the diagnostic of errors.

As a rule of thumb, it is highly recommended to carefully chose the (super)-type annotating the quantified variables in given formula, so as to limit the number of explicit casts. But sometimes, these casts cannot be avoided: notably in the definition of recurrence relations like Equation 3.2, and even more so when these relations have provisos. They involve polynomial expressions in the indices, whose evaluations are rational values. The indices really are integers, and thus have to be casted in the type of rationals when describing the values of these coefficients.

Besides the aforementioned cooperation between a proof assistant and a computer algebra system, a substantial part of the formal development is devoted to the proof of Lemma 3.2. This lemma describes the asymptotic behavior of the sequence $(\ell_n)_{n\in\mathbb{N}}$, of the least common multiple of integers between 1 and n. For this purpose, we have formalized an elementary proof due to Hanson [91], following a suggestion by Alin Bostan.

In Hanson's short note [91], Lemma 3.2 follows from the study of another sequence, defined as a multinomial coefficient from the elements of a fast-growing sequence α. For this sequence, the fact that $\prod_{i=1}^{n} \alpha_i^{1/\alpha_i} < 3$ independently of n then allows to show that $C(n) = \mathcal{O}(3^n)$. More precisely, the sequence $(\alpha_n)_{n\in\mathbb{N}}$ is defined as $\alpha_1 = 2$, and $\alpha_{n+1} = \alpha_1\alpha_2\cdots\alpha_n + 1$ for $n \geq 1$. By an induction on n, this is equivalent to $\alpha_{n+1} = \alpha_n^2 - \alpha_n + 1$. For $n, k \in \mathbb{N}$, let

$$C(n,k) = \frac{n!}{\lfloor n/\alpha_1\rfloor!\lfloor n/\alpha_2\rfloor!\cdots\lfloor n/\alpha_k\rfloor!}.$$

As soon as $\alpha_k \geq n$, $C(n,k)$ is independent of k and we denote $C(n) = C(n,k)$ for all such k. Now the bulk of the proof consists in showing the following majorization for $C(n,k)$:

Lemma 3.5 *Let $k \geq 1$, $n \in \mathbb{N}$. If $\alpha_k \leq n$,*

$$C(n,k) < \frac{n^n \left(\frac{10\,n}{\alpha_1}\right)^{\frac{\alpha_1-1}{\alpha_1}} \left(\frac{10\,n}{\alpha_2}\right)^{\frac{\alpha_2-1}{\alpha_2}} \cdots \left(\frac{10\,n}{\alpha_k}\right)^{\frac{\alpha_k-1}{\alpha_k}}}{\left(\frac{10\,n}{\alpha_1}\right)^{\frac{10\,n}{\alpha_1}} \left(\frac{10\,n}{\alpha_1}\right)^{\frac{10\,n}{\alpha_1}} \cdots \left(\frac{10\,n}{\alpha_k}\right)^{\frac{10\,n}{\alpha_k}}}.$$

The proof of Lemma 3.5 itself combines two ingredients. The first one is an auxiliary majorization of (n,k):

Lemma 3.6 *For $k \geq 1$ and $n \geq 2$,*

$$C(n,k) < \frac{n^n}{\left\lfloor\frac{n}{\alpha_1}\right\rfloor^{\left\lfloor\frac{n}{\alpha_1}\right\rfloor} \cdots \left\lfloor\frac{n}{\alpha_k}\right\rfloor^{\left\lfloor\frac{n}{\alpha_k}\right\rfloor}}.$$

The proof of this first ingredient essentially boils down to properties of multinomial coefficients, like Relation 3.4, which holds for m and the m_i (not all zero) non-negative integers and $m = m_1 + \cdots + m_k$:

$$(m_1 + \cdots + m_k)^m \geq \binom{m}{m_1, \cdots, m_k} m_1^{m_1} \cdots m_k^{m_k}. \tag{3.4}$$

Relation 3.4 is obvious from the fact that for $n, k_1, \cdots, k_l \in \mathbb{N}$ and $n = k_1 + \cdots + k_l$, the multinomial coefficient $\binom{n}{k_1, \cdots, k_l}$ is the coefficient of $x_1^{k_1} \cdots x_l^{k_l}$ in the formal expansion of $(x_1 + \cdots + x_l)^n$. This remark can in fact be taken as a definition of multinomial coefficients. When it is not the case, the corresponding formula is called the *multinomial theorem*. In the formal libraries, $\binom{n}{k_1, \cdots, k_l}$ is defined as the product $\prod_{i=1}^{l} \binom{k_1 + \cdots + k_i}{k_i}$. The main reason for this choice is to provide for free the fact that multinomial coefficients are integers:

```
Definition multinomial (l : seq nat) : nat :=
 \prod_(0 <= i < size l) binomial (\sum_(0 <= j < i.+1) l_j) l_i
```

Note that the argument of the function is just a list of numbers: in the notation $\binom{n}{k_1, \cdots, k_l}$, the sum n is a redundant data, and the index l is the length of the list.

Here is the statement of the multinomial theorem formula, for $n, m \in \mathbb{N}$ and $x_1, \ldots x_s$ in a commutative ring:

$$(x_1 + \ldots x_s)^n = \sum_{t_1 + \ldots t_s = n} \binom{n}{t_1, \ldots, t_s} x_1^{t_1} \ldots x_s^{t_s}$$

Its formal account is quantified over a sequence `l`, representing $x_1, \ldots x_s$, and the formal `\sum_(x <- l)x) ^+ n` is for $(x_1 + \ldots x_s)^n$:

```
Lemma generalNewton (l : seq R) (n m : nat) (s := size l) :
  (n <= m)%N ->
  (\sum_(x <- l) x) ^+ n =
  \sum_(t : s.-tuple I_m.+1 | (\sum_(i <- t) i) == n)
   (multinomial (tmap_val t))%:R * monomial l (tmap_val t).
```

One of the technical issues here lies in the specification of the summation domain in the right hand-side. In the current state of the Mathematical Components library, the big operator [26] used to model the iterated sum can only operates on an explicit, finite sequence of arguments. In our case, this sequence is the list of inhabitants of the finite type of s-tuples of non-negative integers smaller than n

(in fact, and more generally, smaller than any m greater than n). The list of inhabitants of this type `(t : s.-tuple I_m.+1)` is filtered to keep only those that sum to n, i.e. the terms `t` such that `(\sum_(i <- t)i)== n`.

The other additional annotation required here is the function `tmap_val`, which maps a tuple of numbers with bounded values to a list of natural numbers. A term of type `s.-tuple T` is a list of elements of type `T`, paired with a proof that its length is `s`, and can be coerced to the corresponding list by forgetting the proof. A term of type `I_k` is a natural number, paired with a proof that it is smaller than `k`, and can be similarly coerced to the corresponding number. But a term of type `s.-tuple I_k` cannot be coerced to a list of numbers: the function `tmap_val`, which erases both nature of proof annotations, has to be explicitly inserted.

3.5 Numbers and Proofs

This formal proof of Apéry's theorem features several natures of computational steps. The first one is the obvious one: the verification of the certificates generated by the external oracle. The formal proof of Lemma 3.4 relies on the mechanical verification of some algebraic identities, and eventually boils down to the normalization to zero of some polynomials produced by the oracle, plus the verification of some proof obligations phrased in the language of linear arithmetic, coming from manual annotations [47]. The latter are solved instantaneously using Coq's `lia` tactic [28], for linear integer arithmetic. Although the former polynomial expressions can be easily manipulated by a computer-algebra system, their size makes the *interactive* writing of proof scripts quite unusual and challenging.

For example, the oracle guesses a recurrence relation $P \cdot y = 0$ of order four for the sequence $(b_n)_{n\in\mathbb{N}}$ of Equation 3.1. The order of the recurrence is later reduced using initial conditions, so as to verify the order two recurrence of Lemma 3.4. The statement $P \cdot b = 0$, when pretty-printed using Mathematical Components syntax, spans over 8,000 lines of code, and features over 18,600 monomials. On the other hand, the integer coefficients involved in the recurrence are quite small, less that 13 decimal digits. The Maple computer algebra system verifies that $P \cdot b = 0$ in less than 2 seconds. Coq needs a bit less than 4 minutes to produce and check a formal proof. The latter time however includes several ingredients, corresponding to the different phases of the `ring` tactic [87]: the construction of a reified term; the normalization of the corresponding polynomial expression per se; the verification that the reified term is a correct abstract syntax tree for the initial goal. At the time of writing, the first reification phase is unfortunately the most time-consuming, although one would expect it to be neglectible. In fact, this phase can even become the limiting factor on some examples.

This formal proof makes use of a second nature of computations, of smaller scale, akin to resorting to a pocket calculator. Remember that the irrationality of $\zeta(3)$ ultimately follows from the fact that the sequence $\sigma_n = 2\ell_n^3(a_n\zeta(3) - b_n)$ tends to zero while being positive, combined with the fact that if $\zeta(3)$ is a rational number, then σ_n is an integer. Lemma 3.7 can be used to obtain the asymptotic behavior of σ_n can be obtained from the following remark, whose proof is an instance of such smaller scale computations:

Lemma 3.7 $33^n \in O(a_n)$.

Proof Introduce the sequence $\rho_n = a_{n+1}/a_n$ and observe that $\rho_{51} > 33$. We now show that ρ is increasing. Define rational functions α and β so that the conclusion of Lemma 3.4 for a_n rewrites to $a_{n+2} - \alpha(n)a_{n+1} + \beta(n)a_n = 0$ for $n \geq 0$. Now, for any $n \in \mathbb{N}$, introduce the homography $h_n(x) = \alpha(n) - \frac{\beta(n)}{x}$, so that $\rho_{n+1} = h_n(\rho_n)$. Let x_n be the largest root of $x^2 - \alpha(n)x + \beta(n)$. The result follows by induction on n from the fact that $h([1, x_n]) \subset [1, x_n]$ and from the observation that $\rho_2 \in [1, x_2]$. □

Note that the observation that 51 is large enough is best performed outside of Coq, and that computing the value of ρ_{51} is an autarkic computation.

The 33 constant featured by Lemma 3.7 indeed combines well with the estimation of the growth of the sequence ℓ_n, the least common multiple of integers between 1 and n, given by Lemma 3.2. As already mentioned in Section 3.4, our formalization of Lemma 3.2 is based on a concise and elementary proof by Hanson [91]. This proof uses among other things the asymptotic properties of real-valued functions like $x \mapsto (1 + \frac{1}{x})^x$. At the time of writing and up to our knowledge, there is no library about real numbers available in the Coq system that allows to conduct a proof like Lemma 3 in Hanson's note [91] as straightforwardly as it is on paper. The sketch library of real numbers used in this proof was originally designed for the purpose of constructing real algebraic numbers, as a model of real closed field [51]. In particular, it was not designed for the purpose of backing non-elementary proofs in real analysis. It thus lacks a number of elementary arithmetic lemmas, that had to be added to the initial content, in order to be able to work with the equational theory of rational powers of the reals. Moreover, the proof, as phrased in the original paper, makes use of a few transcendental constants and functions, whose formalization would have represented a significant extension of the library with more constructive analysis. We considered several options here: using an existing library about constructive analysis [115], but this one proved not complete enough, and thus not worth the non-trivial plumbing issues; using a non-constructive analysis library [34], but here again the plumbing work to connect data-structures exceeded the benefits. We finally circumvented this issue by

sticking to constructive reals, and devising an even more elementary version of the proof, relying on more small-scale calculations.

What we have just called plumbing represent in fact a recurring modularity issue in our formalization work. Proofs based on evaluations of closed forms formulas put it in the foreground. There is indeed a discrepancy between the data-structures used for a definition of sequences a_n, b_n, ρ_n, etc. that is convenient in proofs, and the ones which are amenable to medium-scale computations, like the upper bound for ρ_{51}. For this purpose, we resorted to the CoqEAL suite of tools [49, 151], which are designed to automate the transfer of computations between equivalent programs operating on different representations of numbers, polynomials, matrices, etc. This approach however still requires writing a significant amount of boilerplate code, in part because the CoqEAL's library seems not enough developed yet. More recent alternatives [163], also based parametricity translations but combined with a univalent approach, might help leveraging this bureaucracy in the future.

3.6 Conclusion and perspectives

Soon after the completion of this proof, Eberl made available a verified proof of Apéry's theorem developed using the Isabelle/HOL proof assistant [66], and based on a short and elegant proof by Beukers [29]. This formalization takes benefit of the more advanced material available for real and complex analysis in the Isabelle/HOL's ecosystem of libraries. This result contributes to a larger collection of formalized chapters of undergraduate number theory, also due to Eberl [67].

The specificity of the proof presented in this chapter is the use of computer algebra methods, which are of a broader interest than this sole irrationality result. In fact, one of the main initial motivations for this work was indeed to provide formally verified automated routines based on ∂-finiteness, so as for instance to prove certain combinatorial identities automatically [143]. Such routines would rely on the cooperation of a computer algebra library (like Maple/Algolib) with a proof assistant, in which the computer algebra system provides candidate datas, which are imported in a proof script and included in a proof which remains independent from how the data have been produced. Two natures of computations are thus involved: a potentially expensive exploratory work, which is performed by the computer algebra systems, e.g. to find a certain recurrence operator, and a verification phase, which amounts to some normalization procedure, e.g. the normalization of a polynomial expression, which is formally verified and performed using the proof assistant. Such an approach is very much in the spirit advocated by John Harrison and Laurent Théry two decades ago [94].

This initial road-map has however been hindered by the unexpected explicit management of provisios required in the cascading verification of recurrence operators for composite ∂-finite sequences. Unfortunately, the corresponding literature is sometimes too vague to justify the correctness of symbolic computations on *actual* multi-indexed sequences (as opposed to germs of sequences). Harrison reports similar shortcomings in his report [93] on the formalization of the Wilf-Zilberger algorithm, which provides a decision procedure for identities involving hypergeometric series. However, he managed in this case to avoid the disgraceful manual treatment of provisos by resorting to a rigidity argument from complex analysis. Unfortunately, we have been so far unable to devise a similar justification for creative telescoping algorithms [48], which are needed for instance to apply to nested sums like $(b_n)_{n\in\mathbb{N}}$.

However, the shortcomings of a few specific algorithms does not alter the importance of ∂-finite functions in the algorithmic study of special functions [111]. More generally the fruitful point of view which consists in representing special functions by equations (and initial conditions) for implementation purposes [156] has a vast and under-used potential in the context of formal verification. The perspectives outlined in Chapter 4 give an example of such possible applications.

Chapter 4

Numerical computations

4.1 Rigorous numerical integration routines

Computing the value of definite integrals is the modern and generalized take on the ancient problem of computing the area of a figure. Quadrature methods hence refer to the numerical methods for estimating such integrals. Numerical integration is indeed often the preferred way of obtaining such estimations as symbolic approaches may be too difficult or even just impossible. Quadrature methods, as implemented in scientific computing systems like, most often consist in interpolating the integrand function by a degree n polynomial, integrating the polynomial and then bounding the error using a bound on the $(n+1)$-th derivative of the integrand function. Estimating the value of integrals can be a crucial part of some mathematical proofs, making numerical integration an invaluable ally. Examples of such proofs occur in various areas of mathematics: Harald Helfgott's proof of the ternary Goldbach conjecture [96] is a prominent such example, in number theory, as well as the first proof of the double bubble conjecture [95], related to the properties of minimal surfaces.

But numerical integration routines, which compute a floating point value for the value of the integral, are prone to subtle issues, and hard to check. Incorrect results can indeed arise from the subtleties of floating-point arithmetics [134], but also possibly from users' ignorance of implicit correctness conditions existing on inputs, typically a regularity assumption needed in the study of the $n+1$-th derivative. Indeed, results are necessarily subject to rounding errors and method errors, which can be studied on paper, but it is difficult to implement a fast routine able to perform a reliable static check that its input verifies the hypothesis of the error analysis. Interval methods [170], also called rigorous numerical methods, compute intervals with floating point endpoints, instead of a single floating-point value and

thus propose a good compromise between reliability and speed. In this Section, we discuss the implementation of formally verified quadrature methods based on interval methods. These implementations are part of the CoqInterval library [130], and thus use its data structures.

4.2 Real numbers, intervals, function extensions

This section contains definitions and notations used in the remainder of the chapter. We call an interval a closed connected subset of the set of real numbers. We use $\mathbb{I}$ to denote the set of intervals: $\{[a;b] \mid a,b \in \mathbb{R} \cup \{-\infty,+\infty\}\}$. A point interval is an interval of the shape $[a;a]$ where $a \in \mathbb{R}$. Any interval variable will be denoted using a bold font. An enclosure of $x \in \mathbb{R}$ is an interval $x \in \mathbb{I}$ such that $x \in \mathbf{x}$. Interval arithmetic is concerned with providing operators on intervals that respect the inclusion property. Given a binary operator $\diamond$ on real numbers, naive interval arithmetic provides a binary operator $\diamondsuit$ on intervals such that:

$$\forall x, y \in \mathbb{R}, \forall \mathbf{x}, \mathbf{y} \in \mathbb{I}, x \in \mathbf{x} \wedge y \in \mathbf{y} \Rightarrow x \diamond y \in \mathbf{x} \diamondsuit \mathbf{y}.$$

There might thus be more clever options, which provide tighter bounds, or more efficient algorithms. In the following, we will not denote interval operators in any distinguishing way. In particular, whenever an arithmetic operator takes interval inputs, it should be understood as any interval extension of the corresponding operator on real numbers. Moreover, whenever a real number appears as an input of an interval operator, it should be understood as any interval that encloses this number. For instance, the expression $(v-u)\cdot\mathbf{x}$ denotes the interval product of the interval $\mathbf{x}$ with any (hopefully tight) interval enclosing the real $v-u$.

For any function $f : \mathbb{R}^n \to \mathbb{R}$, a function $F : \mathbb{I}^n \to \mathbb{I}$ is an interval extension of f on $\mathbb{R}$ if:

$$\forall \mathbf{x_1}, \ldots, \mathbf{x_n}, \quad \{f(x_1, \ldots, x_n) \mid \forall i, x_i \in \mathbf{x_i}\} \subseteq F(\mathbf{x_1}, \ldots, \mathbf{x_n}).$$

Note that this condition is very weak, and that the interval $\{-\infty,+\infty\}$ is a universal, albeit un-informative interval extension. Indeed, this definition will mostly be useful for the purpose of *correctness* theorems: most of the time, one does not even need to assume isotonicity, i.e., that tightening the input of an extension cannot deteriorate its output.

Interval arithmetic trades real numbers for (enclosing) intervals; interval analysis in turn studies interval extensions of functions, so as to model both sources of errors, uncertainty on inputs and approximation methods, and to eventually provide a

correct estimation of the total error. Pioneered by Moore [133] in the mid 60s, interval analysis is the cornerstone of validated numerics [170].

The verified routines presented here make use of a special shape of interval extension, called *rigorous polynomial approximation*. These extensions can be seen as an interval analogue of polynomial approximations, obtained from the truncation of a certain series expansion, for which exact coefficients are replaced by intervals. The truncation induces the method error, and the interval coefficients model the uncertainty sullying the evaluations from which the latter are obtained. For instance, *Taylor models* provide an interval analogue of Taylor expansions. Berz and Makino have advocated the benefits of rigorous polynomial approximations, and Taylor models in particular, in a series of work related to applications in theoretical physics, and in survey articles (e.g., [128]). We denote by $\mathbb{I}[X]$ the set of (finite) tuples of intervals, and, by a slight abuse of notation, we identify every element $(\mathbf{a_n}, \ldots \mathbf{a_0})$ of $\mathbb{I}[X]$ with the univariate interval function $f : \mathbb{I} \to \mathbb{I}$ whose value at $\mathbf{x} \in \mathbb{I}$ is $f(\mathbf{x}) = \mathbf{a_n}\mathbf{x}^n + \cdots + \mathbf{a_1}\mathbf{x} + \mathbf{a_0}$. There exist several variants for the definition of rigorous polynomial approximations; the appropriate definition for our purposes is the following:

Definition 4.1 *For any function $f : \mathbb{R} \to \mathbb{R}$, a* rigorous polynomial approximation *of f on an interval $\boldsymbol{I}$ is a pair $(\boldsymbol{P}, \boldsymbol{\Delta})$ with $\boldsymbol{P} \in \mathbb{I}[X]$ and $\boldsymbol{\Delta} \in \mathbb{I}$ such that there exists a polynomial $P \in \mathbb{R}[X]$ enclosed in $\boldsymbol{P}$ for which:*

$$x \in \boldsymbol{I}, \quad f(x) - P(x) \in \boldsymbol{\Delta}$$

By definition, a polynomial $a_n X^n + \cdots + a_0 \in \mathbb{R}[X]$ is enclosed in a tuple $(\mathbf{a_m}, \ldots, \mathbf{a_0})$ if and only if $n = m$ and for every i, $a_i \in \mathbf{a_i}$.

In the rest of this chapter, we consider two instances of rigorous polynomial approximations, Taylor models and Chebyshev models [107]: note that our definition departs from the one used by Joldes [107] and Makino and Berz [128] by allowing for non-tight interval coefficients. Definition 4.1 however corresponds to the current formalized definition of rigorous polynomial approximations used in the CoqInterval library [130][1].

4.3 Two interval methods for approximating definite integrals

This section summarizes the contributions obtained in publications [122, 123]. The main outcome of this work is an extension of the CoqInterval toolbox [130], which

[1] The present work on quadratures required improving the definition present in CoqInterval at the time of its start.

can approximate proper definite integrals of real univariate functions, of a real variable, and some improper ones. We provide here an overview of the ingredients involved in this formally verified quadrature routine. In the rest of the section we suppose that $F\ \mathbb{I} \to \mathbb{I}$ is an interval extension of the univariate function $f : \mathbb{R} \to \mathbb{R}$, and we want to compute an enclosure of $\int_u^v f$, with $u, v \in \mathbb{R}$ and f integrable on $[u; v]$. For any set $A \subseteq \mathbb{R}$, we denote by $hull(A)$ the *closed convex hull* of A, which is the smallest closed interval containing A. By analogy, for any interval $\mathbf{a} \in \mathbb{I}$, the interval $hull(\mathbf{a}, +\infty)$ is $[inf(\mathbf{a}); +\infty]$.

In order to implement a rigorous quadrature algorithm, a naive albeit natural idea is to devise an interval analogue of Riemann-Darboux sums, and to compute an appropriate partition of the integration interval (e.g., by dichotomy) so as to reach the target precision for the approximation of the integral. The following elementary lemmas are the cornerstone of this algorithm.

Lemma 4.2

$$\int_u^v f \in (v - u) \cdot hull\{f(t) \mid t \in [u; v] \wedge t \in [v; u]\}$$

Proof Easy (see [123]). □

Lemma 4.3 *For any intervals ,***v** *such that* $u \in \mathbf{u}$ *and and* $v \in \mathbf{v}$*, we have:*

$$\int_u^v f \in (\mathbf{v} - \mathbf{u}) \cdot F(hull(\mathbf{u} \cup \mathbf{v}))$$

Proof Easy (see [123]). □

Lemma 4.3 is ultimately used for each interval in the partition, and the results are combined using additivity of integration and interval addition. This lemma motivates a first definition of an interval version of the definition of a coarse interval version of integrals:

```
Variables (f : R -> R) (F : 𝕀.type -> 𝕀.type).

Definition naive_integralprec F u v :=
  𝕀.mul prec (F (𝕀.join u v)) (𝕀.sub prec v u).
```

The Coq formal statement corresponding to Lemma 4.3 provides the corresponding specification to this definition. It can be stated using the additional vocabulary provided by the Coquelicot: library [34], and proved under the assumption that the interval-valued integrand is an interval extension of the real-valued integrand:

```
Hypothesis F_i_extends_f : ∀(xi : 𝕀.type) (x : ℝ),
  (Xreal x) ∈ (𝕀.convert xi) ->
```

```
  (Xreal (f x)) ∈ (𝕀.convert (F xi)).

Lemma naive_integralprec_correct F u v :
(Xreal (RInt f a b)) ∈
         (𝕀.convert (naive_integralprec F u v)).
```

These formal statements are unfortunately slightly complicated by the presence of the `Xreal` and `𝕀.convert` constants. These constants are related to the fact that the containment relation, denoted by the infix $\in$, which relates a real number to an interval containing it, is defined only for *extended* reals, denoted $\overline{\mathbb{R}}$. The type of extended reals completes a copy of the type of real numbers with a bottom element $\bot_{\mathbb{R}}$, so that $\overline{\mathbb{R}} = \mathbb{R} \cup \{\bot_{\mathbb{R}}\}$. This formalization choice is deeply rooted in CoqInterval but it is not essential here.

This enclosure method, using rectangles, is rather crude, but available for a wide class of functions, as the only requirement is the knowledge of an interval extension for the function to be integrated. A better knowledge of the integrated function gives access to higher-order methods, and thus allows for more efficient approaches based on polynomial approximations. In particular, the class of ∂-finite functions mentioned in Chapter 3 allows for an automated computation of Taylor models. Indeed, the linear differential equation that cancels a ∂-finite function induces a recurrence relation relating the coefficients of their Taylor expansion.

The related cornerstone lemma is the following:

Lemma 4.4 *Suppose f is approximated on $[u;v]$ by $p \in \mathbb{R}[X]$ and $\mathbf{\Delta} \in \mathbb{I}$ in the sense that $\forall x \in [u;v], f(x) - p(x) \in \mathbf{\Delta}$. Then for any primitive P of p, we have $\int_u^v f \in P(v) - P(u) + (v-u) \cdot \mathbf{\Delta}$.*

Proof Easy. See (see [123]).

This lemma indicates that integrating rigorous polynomial approximations provides a way to enclose integrals of the approximated function.

4.4 Formally verified approximations of definite integrals

In this section, we give an overview of the additional ingredients required to turn the (formalized versions of the) elementary lemmas in Section 4.3 into a formal-proof-producing quadrature procedure. There are essentially three such additional ingredients: the reification phase, an adaptative dichotomy strategy, and the automated proof of integrability. We will not discuss the dichotomy strategy here, but it is described in our article [123].

The first step performed by the `integral` tactic is the reification phase. It follows the general approach of the CoqInterval library for computing numerical enclosures of real-valued expressions. It consists in checking that the expression e to be bounded belongs to the catalog $\mathcal{E}$ of expressions known to the CoqInterval library. This catalog is the class of expressions recursively built from a single variable and from constants, arithmetic operations, and some elementary functions [130, 123]. As of today, we have:

$$\begin{aligned}\mathcal{E} :=& x \mid \mathbb{F} \mid \pi \mid \\ & \sqrt{\mathcal{E}} \mid \mathcal{E}^k \mid \\ & \mathcal{E}+\mathcal{E} \mid \mathcal{E}-\mathcal{E} \mid \mathcal{E}\times\mathcal{E} \mid \mathcal{E}\div\mathcal{E} \mid -\mathcal{E} \mid ||\mathcal{E}|| \mid \\ & cos(\mathcal{E}) \mid sin(\mathcal{E}) \mid tan(\mathcal{E}) \mid atan(\mathcal{E}) \mid \\ & exp(\mathcal{E}) \mid ln(\mathcal{E})\end{aligned}$$

The next step consists in computing an interval $\mathbf{e}$ such that $e \in \mathbf{e}$ holds *by construction*. Indeed, CoqInterval provides interval operators for each symbol involved the expressions of the catalog, and the inclusion property of interval arithmetic is easily transported from operators to whole expressions by induction on these expressions. Last, the interval $\mathbf{e}$ is compared to the bounds to be established, and if the interval is tight enough, the proof is completed.

In fact, the catalog $\mathcal{E}$ is implemented as a type of abstract syntax trees p, implemented as straight-line programs [130, 123]. For instance, expression $x\ln(1+x)$ could be reified as the following straight-line program, which is meant to operate on an initial stack of values:

```
(* initial stack: [t, 1, 1/4]           *)    Binary Add 1 0
(* current : [t+1, t, 1/4]              *) :: Unary  Ln 1
(* current : [ln(t+1), t+1, t, 1, 1/4] *) :: Binary Mul 1 0
(* current : [t*ln(t+1), t+1, ...]     *) :: nil
```

where `Add`, `Ln` and `Mul` are abstract symbols, labeled with their arity. Note that in this case, the initial stack is larger than needed for this expression only, but would be appropriate for the reification of the complete expression to be bounded.

When provided with a large enough initial stack, a straight light program can be *evaluated*. We denote $[\![p]\!]_{\mathbb{R}}(\overrightarrow{x})$ the result of evaluating the straight-line program p with operators of a real variable, i.e., using the type `R` of real numbers, over an initial stack $\overrightarrow{x}$ of real numbers, that is, a list of terms of type `R`. Similarly, $[\![p]\!]_{\mathbb{I}}(\overrightarrow{\mathbf{x}})$ denotes the evaluation of program p with interval operations and an initial stack $\overrightarrow{\mathbf{x}}$ of intervals.

Then, thanks to the inclusion property of interval arithmetic, we can prove the following formula once and for all:

$$\forall p, \forall \overrightarrow{x} \in \mathbb{R}^n, \forall \overrightarrow{\mathbf{x}} \in \mathbb{I}^n, (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\mathbb{R}}(\overrightarrow{x}) \in [\![p]\!]_{\mathbb{I}}(\overrightarrow{\mathbf{x}}) \tag{4.1}$$

which is the master lemma of the tactic. More precisely, given a goal $A \leq e \leq B$, the tactic first calls an oracle to produce a program p and an initial stack $\overrightarrow{x}$ of real numbers such that $[\![p]\!]_{\mathbb{R}}(\overrightarrow{x}) = e$. Note that the correctness of this oracle does not need to be formally verified; if the oracle fails, then so does the tactic.

The tactic then looks in the context for hypotheses of the form $A_i \leq x_i \leq B_i$, so that it can build a stack $\overrightarrow{\mathbf{x}}$ of intervals such that $\forall i, x_i \in \mathbf{x}_i$. If there is no such hypothesis, the tactic just uses $(-\infty; +\infty)$ for x_i. The tactic can now apply the master lemma 4.1 to replace the goal by $[\![p]\!]_{\mathbb{I}}(\overrightarrow{\mathbf{x}}) \subset [A; B]$. It then attempts to prove this new goal entirely by computation. Note that even if the original goal holds, this attempt may fail due to loss of correlation inherent to interval arithmetic. Formula 4.1 also implies that if a function f can be reified as $t \mapsto [\![p]\!]_{\mathbb{R}}(t, \overrightarrow{x})$, then $t \mapsto [\![p]\!]_{\mathbb{I}}(t, \overrightarrow{\mathbf{x}})$ is an interval extension of f if $\forall i, x_i \in \mathbf{x_i}$. A similar evaluation scheme is used to compute rigorous polynomial approximations for a function f, using the same programs but degree-1 polynomials in the initial stack.

These correctness theorems are also formalized and proved use the Coquelicot library for real analysis [34]. Note that in the classical setting of the Coquelicot library [34], which provides the vocabulary related to integrals that is used here, the type of operator `RInt` is the following:

```
RInt : ∀V : CompleteNormedModule R_AbsRing,
     (R -> V) -> R -> R -> V
```

In particular, for any function `(f : R -> R)` and any real points `(a, b : R)`, the term `(RInt f a b)` is well-formed, whether or not the integrand `f` is integrable on the integration interval. However, when the integral does not exist, no property can be established about this real value. But as expected, and as on paper, the formalized versions of the theorems in Section 4.3 all require that the integrand is integrable on the integration domain. As a consequence, the first task to be performed by the tactic is to produce an integrability proof.

In fact, the catalog $\mathcal{E}$ has a quite specific feature: every function in this catalog is continuous everywhere it is defined; this property can be used to devise a way of producing an integrability proof at little expense. This automation takes benefit of the formalization choices for the type of intervals provided by CoqInterval. It does not only contains pairs of floating-point numbers and half-lines, but also a special

interval $\bot_{\mathbb{I}}$, which is always propagated along computations. An interval operator produces the value $\bot_{\mathbb{I}}$ whenever the input intervals are not fully included in the definition domain of the corresponding real operator. We can thus implement an additional evaluation scheme $[\![p]\!]_{\overline{\mathbb{R}}}(\overrightarrow{x})$ of a program p on extended reals, whose correctness theorem can still be expressed using the type $\mathbb{I}$. This scheme assigns the value $\bot_{\mathbb{R}}$ as soon as an operation is applied to inputs that are outside the usual definition domain of the operator. For instance, the result of dividing one by zero in $\overline{\mathbb{R}}$ is $\bot_{\mathbb{R}}$, while it is unspecified in $\mathbb{R}$. Now we can extend formula 4.1, by generalizing enclosures with the relation $\bot_{\mathbb{R}} \in \bot_{\mathbb{I}}$:

$$\forall p, \forall \overrightarrow{x} \in \overline{\mathbb{R}}^n, \forall \overrightarrow{\mathbf{x}} \in \mathbb{I}^n, (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\overline{\mathbb{R}}}(\overrightarrow{x}) \in [\![p]\!]_{\mathbb{I}}(\overrightarrow{\mathbf{x}}) \tag{4.2}$$

Let us go back to the issue of proving integrability. By definition, whenever $[\![p]\!]_{\overline{\mathbb{R}}}(\overrightarrow{x})$ does not evaluate to $\bot_{\mathbb{R}}$, the inputs $\overline{x}$ are part of the definition domain of the expression represented by p. But we actually have a stronger property: not only is $\overline{x}$ part of the definition domain, it is also part of the continuity domain. More precisely, we can prove the following property:

$$\begin{gathered} \forall p, \forall t_0 \in \mathbb{R}, \forall \overrightarrow{x} \in \mathbb{R}^n, [\![p]\!]_{\mathbb{R}}(t_0, \overrightarrow{x}) \neq \bot_{\overline{\mathbb{R}}} \\ \Downarrow \\ t \mapsto [\![p]\!]_{\mathbb{I}}(t, \overrightarrow{\mathbf{x}}) \text{ is continuous at } t_0 \end{gathered} \tag{4.3}$$

Combining Formulas 4.2 and 4.3, we have that obtaining an informative enclosure for a function on a given interval, that is, one different from $\bot_{\mathbb{I}}$, provides a proof that this function is continuous (and thus integrable) on this interval. Note that this property intrinsically depends on the operations that can appear inside p, i.e., the operations belonging to the class $\mathcal{E}$. Therefore, its proof has to be extended as soon as a new operator is supported in $\mathcal{E}$. In particular, it would become incorrect as such, if the integer part function was ever supported.

4.5 Verified approximations of improper integrals

The `integral` tactic can also compute numerical enclosures of some improper integrals. The latter are computed by splitting the interval into two parts, a proper part which is treated with the previous methods, and the remainder which is handled in a specific way. We thus have to describe how we bound the remainder. We consider improper integrals of the shape $\int_u^v fg$ where either $u = 0$ or $v = +\infty$, and f is bounded. Function g belongs to a catalog of functions with known enclosures of their integral, such as the Bertrand integrals of $x \mapsto x^\alpha \ln^\beta x$, that serve as witnesses of the integrability.

To determine that a certain reminder $\int_u^{+\infty} h$ exists, we have added to Coquelicot a proof of the following Cauchy criterion: this integral exists if and only if for any $v \leq u$, $\int_u^v h$ exists and for all $\varepsilon > 0$, there exists $M > 0$ such that for all $u, v \geq M, |\int_u^v vh| \leq \varepsilon$. We use this criterion to show the following lemma:

Lemma 4.5 *Let $f, g : \mathbb{R} \to \mathbb{R}$. Suppose that, on $[u; +\infty)$, f is bounded, f and g are continuous, and g has a constant sign. Moreover, suppose that $\int_u^{+\infty} g$ exists. Then $\int_u^{+\infty} fg$ exists, and:*

$$\int_u^{+\infty} fg \in \mathit{hull}\{f(t) \mid t \geq u\} \cdot \int_u^{+\infty} g.$$

Proof See [123]. □

The corresponding effective version of this result is:

Lemma 4.6 *Let $F, I_g : \mathbb{I} \to \mathbb{I}$ be interval extensions respectively of f and $x \mapsto \int_x^{+\infty} g$. For any interval $\mathbf{u}$ such that $u \in \mathbf{u}$,*

$$\int_u^{+\infty} fg \in F(\mathit{hull}(u, +\infty)) \cdot I_g(u).$$

In order to use Lemma 4.6, the user need to find a suitable extension I_g for the remainder of the integral of g. In that spirit, the library provides two different useful classes of functions, namely Bertrand integrals, of the form $\int_u^{+\infty} x^\alpha \ln^\beta x$ and integrals of exponential functions, of the form $\int_u^{+\infty} e^{\gamma x}$.

A similar lemma is proved for singularities at 0^+, but Bertrand integrals are the only possible witnesses. Other singular bounds should be first reduced to one of these cases by a preliminary (manual) change of variable.

As a side remark, this treatment of singularities implies that the `integral` tactic knows a few closed forms of integrals, those used in the integrability witnesses scales. As a result, the tactic can prove the following exact formula for a Bertrand integral, which may seem surprising at first sight for a numerical method:

```
Lemma bertrand_eample :
  RInt_gen (fun x => 1 * (powerRZ x 3 * ln x^2))
           (at_right 0) (at_point 1) = 1/32.
Proof.
(* turns equality into two inequalities *)
refine ((fun H => Rle_antisym _ _ (proj2 H) (proj1 H)) _).
(* calls integral, tuning the precision *)
integral with (i_prec 10).
Qed.
```

4.6 A certificate-based approach to numerical enclosures

In this section, we discuss a complementary approach to the numerical enclosure of real-valued functions and integrals, based on the verification *a posteriori* of un-trusted computations, called certificates. In this sense, this approach proceeds from the same idea as the one used to verify recurrence relations in Chapter 3. Indeed, it aims at discharging part of the computation work load involved in some proof to external oracles, while correctness remains guaranteed via subsequent validation steps performed inside the proof assistant. The interest of *a posteriori* validation is also clear in the context of rigorous numerical methods (not necessarily formally verified), which aim at obtaining tight and correct estimations of their error *at the lowest possible cost*. *A posteriori* validation techniques thus consist in reconstructing afterwards an error bound for a candidate approximation. Dating back from the works of Kantorovich about Newton's method, they gained prominence with the rise of modern computers and were applied to numerous functional analysis problems [110, 173, 118, 181]. More recently, those methods were used to compute rigorous polynomial approximations for solutions of linear ordinary differential equations [25, 39]. Broadly speaking, the function to be approximated is characterized as a fixed-point of a contracting operator, from which an error bound is recovered thanks to the Banach fixed-point theorem. Such techniques are thus of special interest for formal verification, as it suffices to formalize the theory about contracting operators and to provide means of computing with those operators. For instance, this general method can be used to produce rigorous and formally-verified Chebyshev approximations of functions on reals, as explained in [40]. The present section provides an outline of this work.

The cornerstone of this family of self-validating methods is the Banach fixed-point theorem, which we formalized as an extension to the Coquelicot library. This library makes an extensive use of *filters* [38, 101] for defining topological concepts. We briefly recall the corresponding terminology in Coquelicot. A filter on a type T is a collection of collections of inhabitants of T which is non-empty, upward closed and stable under finite intersections:

```
Record Filter (T : Type) (F : (T -> Prop) -> Prop) := {
  filter_true : F (fun _ => True) ;
  filter_and : ∀P Q : T -> Prop, F P -> F Q -> F (fun x => P x /\
     Q x) ;
  filter_imp : ∀P Q : T -> Prop, (∀x, P x -> Q x) -> F P -> F Q }.
```

While filters capture the notion of neighborhoods, *balls* allow for expressing the relative closeness of points in the space. Balls are formalized using a ternary rela-

tion between two points in the carrier type, and a real number, with the following axioms:

```
  ball : M -> R -> M -> Prop ;
  ax1 : ∀x (e : posreal), ball x e x ;
  ax2 : ∀x y e, ball x e y -> ball y e x ;
  ax3 : ∀x y z e1 e2,
        ball x e1 y -> ball y e2 z -> ball x (e1 + e2) z
```

Two points are called *close* when they cannot be separated by balls:

```
Definition close (x y : M) : Prop := ∀eps : posreal, ball x eps y.
```

A filter is called a *Cauchy filter* when it contains balls of arbitrary (small) radius:

```
Definition cauchy (T : UniformSpace) (F : (T -> Prop) -> Prop) :=
  ∀eps : posreal, ∃x, F (ball x eps).
```

Finally, a *uniform space* is a type equipped with a ball relation and a *complete space* moreover has a limit operation on filters, which ensures the convergence of Cauchy sequences via the following axioms:

```
  lim : ((T -> Prop) -> Prop) -> T ;
  ax1 : ∀F, ProperFilter F -> cauchy F -> ∀eps : posreal, F (ball
     (lim F) eps) ;
  ax2 : ∀F1 F2, F1 ⊆ F2 -> F2 ⊆ F1 -> close (lim F1) (lim F2)
```

(where `ProperFilter F` is equivalent to `Filter F /\ ∀P, F P -> ∃x, P x`).

We now start describing our extension of the library. The above formal definition of balls does not enforce closedness nor openness. We thus introduced the relation associated with the *closure* of balls, so as to model *closed* neighborhoods:

```
Definition cball x r y := ∀e : posreal, ball x (r+e) y.
```

Equipped with this definition, hypothesis (ii) of Theorem 4.7 is formalized as follows:

```
Definition lipschitz_on (F : U -> U) (mu : R) (P : U -> Prop) :=
∀x y : U, ∀r : nonnegreal, P x -> P y -> cball x r y -> cball (F x
   ) (mu*r) (F y).
```

Now let us recall the statement of the Banach fixed-point theorem. We use a slightly non-standard phrasing which is well-suited to our context.

Theorem 4.7 (Banach fixed-point) *Let $(X, \|\cdot\|)$ be a Banach space, an operator $\boldsymbol{F} : X \to X$, $h^\circ \in X$, and $\mu, b, r \in \mathbb{R}_+$, satisfying the following conditions:*

(i) $\|h^\circ - \boldsymbol{F} \cdot h^\circ\| \leq b$;

(ii) $\boldsymbol{F}$ *is* μ*-Lipschitz over the ball* $\overline{B}(h^\circ, r) := \{h \in X \mid \|h - h^\circ\| \leq r\}$*;*

(iii) $\mu < 1$ *:* $\boldsymbol{F}$ *is contracting over* $\overline{B}(h^\circ, r)$*;*

(iv) $b + \mu r \leq r$*.*

Then $\boldsymbol{F}$ *admits a unique fixed-point* h^* *in* $\overline{B}(h^\circ, r)$*.*

We now sketch our formalized proof, using mathematical notations. We consider a complete space X and we write $y \in B(x, r)$ for the formal `(ball x r y)`, and $y \in \overline{B}(x, r)$ for `(cball x r y)`. The key notion is that of *strongly stable ball*:

Definition 4.8 (Strongly stable ball) *A ball* $\overline{B}(u, r)$ *is* μ-strongly stable *for* $\boldsymbol{F}$ *if* $\boldsymbol{F}$ *is* μ*-Lipschitz on* $\overline{B}(u, r)$ *and if there is a non-negative real number* s*, called the* offset*, s.t.:*

$$\boldsymbol{F} \cdot u \in \overline{B}(u, r) \qquad \textit{and} \qquad s + \mu r \leq r.$$

Remark 4.9 (Stability) *For any* x *in* $\overline{B}(u, r)$*, a strongly stable ball for* $\boldsymbol{F}$*,* $\boldsymbol{F} \cdot x \in \overline{B}(u, r)$*.*

Remark 4.10 (Contracting case) *When* $0 \leq \mu < 1$*, for any* μ*-strongly stable ball* $\overline{B}(v, \rho)$*, with offset* σ*,* $\overline{B}(\boldsymbol{F} \cdot v, \mu\rho)$ *is also a strongly stable ball, with offset* $\mu\sigma$*. Moreover,* $\overline{B}(\boldsymbol{F} \cdot v, \mu\rho)$ *is included in* $\overline{B}(v, \rho)$*.*

Assume that $\boldsymbol{F}$ has a μ-strongly stable ball $\overline{B}(u, r)$ of offset s, with $\mu < 1$. In particular, $\boldsymbol{F}$ is contracting on $\overline{B}(u, r)$. Consider the sequence of balls defined as follows:

$$\overline{B}_n = \overline{B}(u_n, r_n) \quad \text{with} \quad u_n = \boldsymbol{F}^n \cdot u \quad \text{and} \quad r_n = r\mu^n$$

where $\boldsymbol{F}^n \cdot u$ denotes the iterated images of u under $\boldsymbol{F}$. By Remark 4.10, $(\overline{B}_n)_{n \in \mathbb{N}}$ is a nested sequence of μ-strongly stable ball for $\boldsymbol{F}$, with offset $s\mu^n$. Let $\mathcal{F}$ be the family of collections of points in U defined as:

$$\mathcal{F} = \{P \subseteq U \mid \exists\, n, \overline{B}_n \subseteq P\}.$$

It is a proper filter: $\mathcal{F}$ contains U, it is obviously upward closed, and for $P, Q \in \mathcal{F}$, $P \cap Q$ is also in $\mathcal{F}$ because $(\overline{B}_n)_{n \in \mathbb{N}}$ is decreasing for inclusion. Thus $\mathcal{F}$ has a limit w, such that for any $\varepsilon > 0$, balls $\overline{B}_n$ are eventually included in $B(w, \varepsilon)$. We provide a formal proof of Theorem 4.11, a reformulation of Theorem 4.7 using the vocabulary of the Coquelicot library:

Theorem 4.11 *The limit* w *of the filter* $\mathcal{F}$ *is in* $\overline{B}_0$*, and* w *is a fixed point of* $\boldsymbol{F}$*. Moreover,* w *is close to every other fixed point of* $\boldsymbol{F}$ *in* $\overline{B}_0$*.*

Proof In this statement "w is a fixed point of $\boldsymbol{F}$" means "w is close to $\boldsymbol{F} \cdot w$". First, $w \in \overline{B}_n$ for all n. Indeed, for any $\varepsilon > 0$, there is an $m \geq n$ s.t. $\overline{B}_m \subseteq B(w, \varepsilon)$, and since $\overline{B}_m \subseteq \overline{B}_n$, $u_m \in \overline{B}_n \cap B(w, \varepsilon)$. In particular, $w \in \overline{B}_0$. It is also clear by stability that $\boldsymbol{F} \cdot w \in \overline{B}_n$ for all n. Moreover, w is close to any point v s.t. $v \in \overline{B}_n$ for all n (for any $\varepsilon > 0$, choose n s.t. $2\mu r^n < \varepsilon$). Taking $v := \boldsymbol{F} \cdot w$ proves that w is a fixed point of $\boldsymbol{F}$.

Finally, if $w' \in \overline{B}_0$ is another fixed point of $\boldsymbol{F}$, then it follows from an easy induction that $w' \in \overline{B}_n$ for all n. Hence, the foregoing shows that w is close to w'. □

Strongly stable balls model the requirements set on the untrusted data to be formally verified. They can also be seen as balls centered at the initial point, and large enough to include all its successive iterates, i.e. as instances of the locus at stake in standard presentations of the proof. This result has been formalized in various flavors of logic and of proof assistants. The version proved by Boldo et al. [33], also on top of the Coquelicot library, has a slightly more technical wording, which seems to be made necessary by its further usage in the verification of the Lax-Milgram theorem. Our proof script is significantly shorter, partly because we automate proofs of positivity conditions (for radii of balls) using unification hints for manifestly positive expressions. But the key ingredient for concision is to make most of the filter device in the proof, and to refrain from resorting to low-level properties of geometric sequences. To the best of our knowledge, the other libraries of formalized analysis featuring a proof of this result, notably in the Isabelle/HOL and HOL-Light systems, are based on variants of proof strategy closer to the approach of Boldo et al. than to ours.

We now present in more detail the general principle of fixed-point-based *a posteriori* validation methods, and more particularly, the use of Newton-like validation operators.

Throughout this section, let $(X, \|\cdot\|)$ denote a Banach space, and h^* the exact solution of an equation in X. In this article, X stands for the space $\mathcal{C}(I)$ of real-valued continuous functions defined over a compact segment $I = [a, b]$, with the uniform norm $\|h\| := \sup_{x \in I} |h(x)|$. The division and square root of functions are simple examples of solutions of equations in $\mathcal{C}(I)$, but there are also differential equations, integral equations, delay equation, etc. The general scheme for Banach fixed-point based *a posteriori* validation methods follows two steps:

1. **Approximation step.** A numerical approximation $h^\circ \in X$ of h^* is obtained by an oracle, which may resort to any approximation method. In particular, this step requires no mathematical assumption and can be executed purely numerically outside the proof assistant, good approximation properties being only desirable for efficiency. In our setting, with $X = \mathcal{C}(I)$, we used

interpolation at Chebyshev nodes, which provides an efficient and accurate oracle.

2. **Validation step.** The initial problem is rephrased in such a way that h^* is a fixed point of a (locally) contracting operator $\boldsymbol{F} : X \to X$. An *a posteriori* error bound on $\|h^\circ - h^*\|$ is deduced from the Banach fixed-point theorem (Theorem 4.7).

We thus need to find a contracting operator $\boldsymbol{F}$ of which h^* is a fixed point. To this end, we use Newton-like validation methods, which transform an equation $\boldsymbol{M} \cdot h = 0$ into an equivalent fixed-point equation $\boldsymbol{F} \cdot h = h$ with $\boldsymbol{F}$ contracting. More specifically, suppose that $\boldsymbol{M} : X \to Y$ is differentiable; we use a Newton-like operator $\boldsymbol{F} : X \to X$ defined as:

$$\boldsymbol{F} \cdot h = h - \boldsymbol{A} \cdot \boldsymbol{M} \cdot h, \qquad h \in X,$$

with $\boldsymbol{A} : Y \to X$ an *injective bounded linear* operator, intended to be close to $(\mathcal{D}\boldsymbol{M}_{h^\circ})^{-1}$. The operator $\boldsymbol{A}$ may be given by an oracle and does not need to be this exact inverse, which anyway might be non representable on computers exactly. The mean value theorem yields a Lipschitz ratio μ for $\boldsymbol{F}$ over any convex subset S of X:

$$\forall h_1, h_2 \in S,\ \|\boldsymbol{F} \cdot h_1 - \boldsymbol{F} \cdot h_2\| \leq \mu \|h_1 - h_2\|, \quad \text{with } \mu = \sup_{h \in S} \|\mathcal{D}\boldsymbol{F}_h\| = \sup_{h \in S} \|\mathbf{1}_X - \boldsymbol{A} \cdot \mathcal{D}\boldsymbol{M}_h\|,$$

which is expected to be small over some neighborhood of h°.

Concretely, in order to apply Theorem 4.7, one needs to compute the following quantities:

- a bound $b \geq \|\boldsymbol{A} \cdot \boldsymbol{M} \cdot h^\circ\| = \|h^\circ - \boldsymbol{F} \cdot h^\circ\|$;
- a bound $\mu_0 \geq \|\mathbf{1}_X - \boldsymbol{A} \cdot \mathcal{D}\boldsymbol{M}_{h^\circ}\| = \|\mathcal{D}\boldsymbol{F}_{h^\circ}\|$;
- a bound $\mu'(r) \geq \|\boldsymbol{A} \cdot (\mathcal{D}\boldsymbol{M}_h - \mathcal{D}\boldsymbol{M}_{h^\circ})\| = \|\mathcal{D}\boldsymbol{F}_h - \mathcal{D}\boldsymbol{F}_{h^\circ}\|$ valid for any $h \in B(h^\circ, r)$, and parameterized by a radius $r \in \mathbb{R}_+$.

If we are able to find a radius $r \in \mathbb{R}_+$ satisfying:

$$\mu(r) := \mu_0 + \mu'(r) < 1, \qquad \text{and} \qquad b + r\mu(r) \leq r, \tag{4.4}$$

then Theorem 4.7 guarantees the existence and uniqueness of a root h^* of $\boldsymbol{M}$ in $B(h^\circ, r)$.

Remark 4.12 *Finding an r as small as possible while satisfying (4.4) may be an nontrivial task for automated validation procedures. For many problems, $\mu'(r)$ is polynomial, hence conditions (4.4) are polynomial inequalities over r: this is called the* radii polynomial approach *[102] in rigorous numerics.*

Finally here are two concrete examples of validation steps, for the division operation, and for square root. These cases are rather elementary as division (resp. square root) induces an affine (resp. quadratic) equation, which admits closed form solutions.

For $f, g \in \mathcal{C}(I)$ with g non-vanishing over I, the quotient f/g is the unique root of $\boldsymbol{M} : h \mapsto gh - f$. Let h° be a candidate approximation given by the approximation step. Constructing the Newton-like operator $\boldsymbol{F}$ requires an approximation $\boldsymbol{A}$ of $(\mathcal{D}\boldsymbol{M}_{h^\circ})^{-1} : k \mapsto k/g$. For that purpose, suppose $w \approx 1/g \in \mathcal{C}(I)$ is also given by an oracle, and define:

$$\boldsymbol{F} \cdot h = h - w(gh - f). \tag{4.5}$$

The next proposition computes an upper bound for $\|h^\circ - f/g\|$; its formalization in Coq provides the desired validation step.

Proposition 4.13 *Let $f, g, h^\circ, w \in \mathcal{C}(I)$, and $\mu, b \in \mathbb{R}_+$ such that:*

$$\|w(gh^\circ - f)\| \le b, \quad (4.13\,\text{i}) \qquad \|1 - wg\| \le \mu, \quad (4.13\,\text{ii}) \qquad \mu < 1. \quad (4.13\,\text{iii})$$

Then g does not vanish over I and $\|h^\circ - f/g\| \le b/(1-\mu)$.

Proof Conditions (4.13 ii) and (4.13 iii) imply that $\boldsymbol{F}$ (Equation (4.5)) is contracting over $\mathcal{C}(I)$ with ratio μ. The radius $r := \frac{b}{1-\mu}$ makes the ball $\overline{B}(h^\circ, r)$ strongly stable with offset b (4.13 i), since $b + \mu r = r$. Therefore, h^* is the (global) unique root of $\boldsymbol{M}$, and $\|h^\circ - h^*\| \le r$.

Finally, w and g do not vanish because $\|1 - wg\| \le \mu < 1$. Hence, $h^* = f/g$ over I. □

Let $f \in \mathcal{C}(I)$ be strictly positive over I. The square root $\sqrt{f}$ is one of the two roots of the quadratic equation $\boldsymbol{M} \cdot h := h^2 - f = 0$ (the other being $-\sqrt{f}$). Let h° be a candidate approximation. Since $\mathcal{D}\boldsymbol{M}_h : k \mapsto 2hk$, one also needs an approximation $w \approx 1/(2h^\circ) \approx 1/(2\sqrt{f}) \in \mathcal{C}(I)$ in order to define $\boldsymbol{A} : k \mapsto wk$, approximating $(\mathcal{D}\boldsymbol{M}_{h^\circ})^{-1}$. Then:

$$\boldsymbol{F} : \quad h \quad \mapsto \quad h - w(h^2 - f). \tag{4.6}$$

The next proposition computes an upper bound for $\|h^\circ - \sqrt{f}\|$: its implementation in Coq provides the corresponding validation step.

Proposition 4.14 *Let $f, h^\circ, w \in \mathcal{C}(I)$, $\mu_0, \mu_1, b \in \mathbb{R}_+$ and $t_0 \in I$ such that:*

$$\left\| w\left(h^{\circ 2} - f\right)\right\| \leq b, \quad (4.14\,\text{i}) \qquad \|1 - 2wh^\circ\| \leq \mu_0, \quad (4.14\,\text{ii}) \qquad \|w\| \leq \mu_1, \quad (4.14\,\text{iii})$$

$$\mu_0 < 1, \quad (4.14\,\text{iv}) \qquad (1-\mu_0)^2 - 8b\mu_1 \geq 0, \quad (4.14\,\text{v}) \qquad w(t_0) > 0. \quad (4.14\,\text{vi})$$

Then $f > 0$ over I and $\left\|h^\circ - \sqrt{f}\right\| \leq r^*$, where:

$$r^* := \frac{1 - \mu_0 - \sqrt{(1-\mu_0)^2 - 8b\mu_1}}{4\mu_1}.$$

Proof First, since $\|1 - 2wh^\circ\| \leq \mu_0 < 1$ (by (4.14 ii) and (4.14 iv)) and $w(t_0) > 0$ (4.14 vi), w and h° are strictly positive over I, by continuity. Using (4.14 iii), $\mu_1 > 0$.

If $b = 0$, then $r^* = 0$ and $h^\circ = \sqrt{f}$ over I, because $w(h^{\circ 2} - f) = 0$ (4.14 i) and $w, h^\circ > 0$. Hence the conclusion holds.

From now on, we assume $b > 0$. $\boldsymbol{F}$ is Lipschitz of ratio $\mu(r) := \mu_0 + 2\mu_1 r$ over $\overline{B}(h^\circ, r)$ for any $r \in \mathbb{R}_+$, because:

$$\begin{aligned}\boldsymbol{F}\cdot h_1 - \boldsymbol{F}\cdot h_2 &= (h_1 - h_2) - w(h_1^2 - h_2^2)\\ &= \left[(1 - 2wh^\circ) + w(h^\circ - h_1) + w(h^\circ - h_2)\right](h_1 - h_2).\end{aligned}$$

Therefore, satisfying $b + \mu(r)r \leq r$ is equivalent to the quadratic inequality:

$$2\mu_1 r^2 + (\mu_0 - 1)r + b \leq 0. \tag{4.7}$$

Condition (4.14 v) implies that (4.7) admits solutions, and r^* is the smallest one. Moreover, since $b, \mu_1 > 0$, we get $r^* > 0$, so that $b + \mu(r^*)r^* = r^*$ also implies $\mu(r^*) < 1$.

Now, all the assumptions of Theorem 4.7 are fulfilled. Hence, $\boldsymbol{F}$ has a unique fixed point h^* in $\overline{B}(h^\circ, r^*)$. To obtain $h^* = \sqrt{f}$ over I, it remains to show that $h^* > 0$. This follows from $w > 0$ and:

$$\|1 - 2wh^*\| \leq \|1 - 2wh^\circ\| + \|2w(h^* - h^\circ)\| \leq \mu_0 + 2\mu_1 r^* = \mu(r^*) < 1. \qquad \square$$

Remark 4.15 *The results in this section have been presented with a classical mathematics bias. However, there is no difficulty in rephrasing them into constructive variants, that would justify the same numerical enclosures.*

4.7 Conclusion and perspectives

Rigorous numerical methods is an active research domain today which covers both fundamental algorithmic contributions and full-fledged libraries providing efficient implementations of the latter algorithms, themselves based on fast interval (or ball) arithmetics. Tucker's proof of the strangeness of the Lorentz attractor [169], based on a rigorous solver for ordinary differential equations, is probably one of the most famous achievements in this area.

However, the experiments we carried in order to benchmark the `integral` tactic described in Section 4.3 have revealed a few shortcomings in numerical routines representative of the state of the art. Notably, these benchmarks illustrated on a few examples how some quadrature routines in Octave [65] and INTLAB [152] can provide off results without warning. The `interval` tactic also showed that one of the bounds used in Helfgott's first generation proof of the ternary Goldbach conjecture, obtained with VNODE-LP [137], was actually incorrect. See [123] for the detailed benchmarks.

Tracking statically the necessary assumptions on entries that guarantee the correctness of enclosures, without compromising its efficiency is more often than not an unsolvable dilemma. Rigorous *and* formally verified numerical routines thus have a clear interest, despite their worse efficiency. However, as of today, the available formally verified rigorous numerics toolbox remains quite incomplete. Besides Guillaume Melquiond's CoqInterval library, which this work extends, a remarkable achievement is the formally verified solver of ordinary differential equations implemented by Fabian Immler using the Isabelle/HOL proof assistant, which he used in particular to study the Lorentz attractor [104].

The `integral` tactic described in this chapter can (and should) be extended in a number of directions. Such extensions would roughly fall in two camps: first, improving the efficiency of the procedure, either by implementing new algorithms, or by using new techniques; second, extending the skills of the procedure, so as to be able to deal with more enclosure problems. We mention only a few of them. For instance, the efficiency of the procedure owes a lot to its adaptative heuristic for computing a suitable partition of the integration domain, together with the appropriate approximations of the integrand, one per each sub-interval. Currently, this heuristic is implemented using Coq, and it is run by the internal code of the tactic. In order to improve efficiency, an external oracle could provide a template for the tactic, which could always be refined afterwards, so as to cut unnecessary intermediate calculations as much as possible. Regarding the extensions of the skills of the procedure, there are again several possible lines of action. For instance, the catalog of functions the tactic can deal with could be significantly extended by

introducing some support for ∂-finite functions, for which efficient approximation algorithms exist that apply uniformly to all members of the class. An extension for supporting complex-valued functions, based on ball arithmetic, would also be very useful in many applications, including for instance other estimations in Helfgott's proof of the ternary Goldbach conjecture, currently obtained with the Arb library [105]. Finally, extending the procedure so as to be able to deal with solutions of more general differential, or functional, equations represents a vast amount of possibilities. See for instance Warwick Tucker's introductory text [170].

Chapter 5

Perspectives

This last chapter is devoted to the medium to long term research directions that I would like to investigate in the few coming years. This research program consists, broadly speaking, in improving the reliability of computer-produced mathematics. More precisely, its central objects of study will be symbolic methods, the substrate of their implementation, and the verification of some of its recent applications in computational number theory.

5.1 Context

A large part of computer *proofs* today are essentially produced by symbolic computations. At the core of computer algebra are exact data such as (unbounded) integers, rational numbers, polynomials, matrices. But symbolic methods also exist in approximation theory, as illustrated in Chapter 4: in this case, routines trade hardly achievable exact results for reliable numerical computations, e.g., intervals with floating-point endpoints instead of real numbers, thus making it possible to endow their approximate results with explicit error bounds.

State-of-the-art symbolic methods today are fast. For instance, it takes less than 10 hours to break 512-bit RSA keys on Amazon EC2 (for less than $100), hence the threat of the Factoring RSA Export Keys (FREAK) attack, on the deliberately weak export cipher suites introduced under the pressure of US government agencies [1]. And fast symbolic methods are available for exploring the properties of a variety of mathematical objects. For instance, high-performance, rigorous, symbolic-numeric methods are used, e.g., to design large particle accelerators [128], or in space applications [157].

Computer algebra systems are the main vehicle of symbolic methods. Besides

the libraries implementing cutting-edge fast algorithms, these complex systems are also powered by a domain specific language, used to formulate queries, and sophisticated graphical interfaces, which allow to produce documents interleaving text and graphics, with calculations. The major ones, like the commercial, closed-code Mathematica, Maple, and the free, open-source, SageMath are fast, multi-purpose, and easy to use. They are thus used both as research tools and in the classroom. Matlab and Octave, leading scientific computing software, can also be extended with symbolic-numeric toolboxes such as INTLAB, which provides rigorous numerical algorithms of about the same efficiency as the fastest pure floating-point routines, using the fastest compilers available [152]. Some more specialized tools provide a portfolio of algorithms of special interest in some identified areas. For instance in number theory, the Pari/GP system provides state-of-the-art computational algebra [140], while Arb provides state-of-the-art symbolic-numeric algorithms, based on ball complex arithmetic [105].

The material presented in Chapter 4 however provides a first illustration that computer algebra is a giant with a feet of clay. We reported the problems we identified to INTLAB developers; unfortunately, they could only fix the bug by dropping support for the absolute value [153]. This is a well-known pitfall: state-of-the-art libraries in scientific computing, whether numerical or purely symbolic, cannot afford tracking the regularity assumptions that would ensure the correctness of their results. Implementations of symbolic methods today are deliberately geared towards speed, at the price of correctness. In the best case, the correctness assumptions of a given routine are documented, but more often than not, they remain implicit, and hidden to end users. For instance, the documentation of the solver of differential equations implemented in the Wolfram Language (Mathematica) only states that "*the answer might not be valid for certain exceptional values of the parameters*" [179]. But it does not explain which ones.

This goes beyond the traditional definition of a bug as a programming error. In fact, one could even go as far as saying that computer algebra systems and their numerical extensions are *wrong by design*.

In this sorry state of affairs, it is all the more worrying that no explicit policy exists for auditing software that produce proof steps in submitted papers, even in high-profile mathematical journals. Generally speaking, the social process of peer-reviewing just falls short of evaluating the proofs produced by computers, as reported by Hales after the publication of his computer-aided proof of the Kepler Conjecture [16].

A diverse range of verification methods, however, exist for auditing software and assess its dependability. Logic- and computer-based formal methods have been on the rise, motivated by the huge money losses, as well as human deaths, that bugs

have caused. Among them, proof assistants provide the most versatile approach. They can be used to represent sophisticated constructions of cutting-edge research in mathematics, like the perfectoid spaces introduced by Schölze in 2012 [41], and to verify large and complex proofs, like the odd order theorem discussed in Chapter 2. They can also verify large-scale programs such as an operating system [112], or a realistic compiler for the C language [117]. They can even generate state-of-the-art low-level code for fast cryptographic arithmetic [70]. However in practice, almost no software cited in publications in mathematics is verified.

Quoting again the Wolfram Language documentation, *"treating expressions like $f[x]$ as both symbolic data and the application of a function f provides a uniquely powerful way to integrate structure and function"* [180] and it is in fact a central design decision to computer algebra systems. Sadly, this integration blurs the interaction between the simplification rules that normalize symbolic data, and the reduction rules that govern the evaluation of programs. For a symbolic expression $E(\alpha, \beta)$, with two free variables α and β, it may happen that substituting first α for β, then β by -1 leads to a *different* result than substituting both α and β by -1, as illustrated by Johansson [106] using Mathematica but also SymPy, the fundamental package for symbolic mathematics in Python. We replay it in Listing 5.1:

```
1 >>> simplify(hyper([n],[m],x).subs({m:-1, n:-1, x:1}))
2                                                        2
3 >>> simplify(hyper([n],[m],x).subs(m, n)).subs({n:-1, x:1})
4                                                        E
```

Listing 5.1: Substitutivity does not hold in computer algebra systems

His example involves the expression `hyper([n],[m],x)`, with three free variables `n`, `m`, and `x`; it represents the (generalized) hypergeometric function ${}_1F_1(n; m; x)$. In a first query, at line 1, symbolic variables `m` and `n` are both substituted by `-1`, and variable `x` by `1`. In a second query, at line 3, `m` is first substituted by the symbolic variable `n`; the expression now no more depends on `m`, and variable `n` is substituted by `-1`, and `x` by `1`. In both cases, the resulting expressions from these substitutions are simplified, and this yields to completely different results: the integer `2` in the first case, and `E`, the base of the natural logarithm, in the second. What happens here is of a similar nature as the shortcomings of creative telescoping algorithms discussed in Chapter 3.

Users simply have no control on the simplification strategy implemented by computer algebra systems, which may use incorrect rules by ignoring provisos, and confuse abstractions. This assessment actually motivated the work presented in

Chapter 3, so as to provide a more reliable version of Salvy's existing Maple worksheet [155]. Such severe design flaws prevent the design of any useful semantics, a first step towards specification and verification, and ruin the hope for *a posteriori* verification approaches, e.g. using Hoare-logic like assertions. In some luck cases, verifying results post hoc, without any assumption on the way data have been produced, is of course possible with a minimal amount of trusted computation, when a certificate-based method exists: Chapter 3 and Chapter 4 provide two non-trivial examples of such a situation. But verifying certificates demands a fast trusted verifier, which is often itself a computer algebra routine.

In fact, such design flaws even compromise the meaning of tests, and forbid the use of such domain specific languages in any critical application. As expected, it also leads to wrong proofs. For example, one of the verified quadrature routines presented in Chapter 4 was used by my co-author Florent Bréhard to invalidate a proved bound on a quantity related to Hilbert's 16th problem, and to compute a new (at the time) record bound, stamped with a Coq formal proof [40].

The formidable advances of symbolic computation has led to high-speed but unreliable implementations but the sources of unreliability are of a fundamental nature.

5.2 Positioning

"Are we just getting wrong answers faster?" Stadtherr's unkind question [162] to the community of high-performance computing not only remains valid today, but it extends to the realm of computer algebra. While machine-assisted design and verification have become standard for critical systems or when security and privacy issues are at stake, only makeshift techniques are available for auditing computer-produced mathematics, far behind the current state of the art in program verification.

The main challenge posed by the verification of computer-produced mathematics is the sophisticated vocabulary required in the specifications of the corresponding code. Verifying computer algebra in the large is demanding, as elementary specifications will casually involve quantifying over objects such as "finite fields of an arbitrary characteristic p", with a formal integer p. Such a parameterization is typically beyond the skills of computer algebra systems; they only provide concrete instances of these fields, for concrete values of p, as this prime integer controls algorithmic choices in modular arithmetic. In fact, verifying computer algebra calls for the most expressible logic a verification tool can be based on: dependent type theory.

Computer algebra systems and proof assistants are both designed with the same

ambition: doing mathematics by computers. But they are developed by disjoint communities, with different motivations: computer algebra systems seek speed when proof assistants enforce correctness. The objective of this research program is to reconcile these two approaches to computer-produced mathematics, and the corresponding research areas.

However, retrofitting correctness in systems that have sacrificed semantics for speed is not an option, therefore, I propose build a new nature of mathematical software, grounded in the rigorous foundations of a proof assistant, Coq [54]. But in turn, catching up on efficiency requires radical changes in the programming environment of dependently typed proof assistants, while remaining compatible with their logical foundations.

The ultimate goal of my research program can thus been seen as the materialization of a verified compilation scheme from textbook formulas to machine code, with feedback to formal proofs.

Figure 5.1 illustrates this approach on the example of estimating $\sqrt{\pi}$, a computational step in a larger mathematical proof. Successive translations refine the various views implicitly overlaid on the expression, in the course of the computer proof. Using Coq's higher-order, dependently typed, programming language it is possible to express all these views, to specify and to verify each transformation phase, and to reuse the value, in a broader formal proof context.

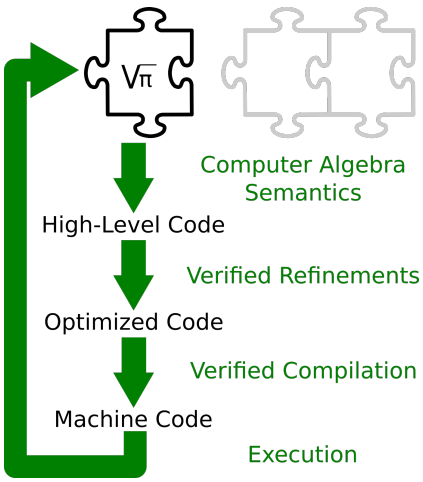

Figure 5.1: From formula to formally verified computation, and back

5.3 Methodology

This program would consist in investigating three scientific objectives, presented below together with the corresponding envisioned methodology.

5.3.1 Achieve high-level syntax and fast low-level, in a proof assistant.

This objective addresses the design of the programming environment underpinning our software. Finding the right balance between strong typing and productivity is a long standing open question in the design of scripting languages. The challenge is to erase the overhead of logic so as to generate efficient code, which can still be reused in formal proofs, without compromise on their trustworthiness. This challenge is compounded by the need to serve two kinds of users [74]: end users, who want a high-level language that makes it simple and fast to state their problem and the way to solve it, and developers, who need to get their computationally intensive programs as close as possible to the bare metal of the machine.

Design a high-level, interpreted, domain-specific language for computer algebra. Gallina, the programming language provided by Coq, has higher-order pure functions and dependent types. Operating on syntax trees, at the core of computer algebra, is a sweet spot for the sophisticated pattern-matching features of Gallina. But at the same time, Gallina blatantly lacks essential meta-programming features, like quotations and advanced binder management. As a programming language, it also desperately lacks some pure features, like exceptions and generalized recursion, as well as impure ones: fresh name generation, timer, file access, etc. As Coq currently stands, the best approach to these imperative extensions seems to be a monadic encoding [127]. This powerful mechanism to capture side effects at large is at the core of Haskell, a leading pure functional programming language, or of F*, used in security-oriented realistic applications. However, monadic programming is hopelessly alien to the community of computer algebra. The design of a neat surface language, which can still be adequately interpreted and optimized by Coq, is an outstanding challenge.

Provide a deeply embedded layer of verified low-level constructs. The efficiency of a state-of-the-art linear algebra toolbox is tied to fine-tuned, in-place operations on low-level data structures, e.g., imperative arrays and pointers. With the help of expert colleagues, we would thus design a low-level language well-suited to the verification of code that relies in a crucial way on actual low-level features, for efficiency reasons. It will come with a formalized operational semantics, close to C, and it will use dependent types to annotate programs, so as to ease subsequent verification. By contrast with other verification-oriented low-level languages [144],

it will be deeply embedded in Gallina, and meant to be compiled to machine code, using a verified compilation scheme [117], before being executed. We ambition to go beyond the state of the art in verified low-level code [164], by reaching the same level of guarantee, but for an integrated language to a higher-level layer, which can in particular be used to write code generators.

5.3.2 Powerful verification frameworks

The complete verification of symbolic methods in a foundational proof assistant, and in particular, its automation, poses specific research problem. Beyond the traditional acceptation of trusted decision procedures, this objective is about automation based on the study of the structural properties of programs, of data structures, and of mathematical objects.

Verified refinements. Verified refinements [4, 5, 116] usually consider step-wise tool-chains for connecting higher-level representations of programs, typically inside the logic implemented by the proof assistant, to some generated code, meant to be executed. Here, we need to extend both ends of the chain. At the top end, principled tools could refine an *existence theorem* (primality is decidable) into an efficient program (a primality test), by refining both algorithms and data structures, and by justifying formally the successive steps of the transformation. At the bottom, automation tools [43] would verify code generators for the low-level language of the system, written using the high-level language from in the first objective. The suite of features provided by both languages is challenging, but the fact that the entire chain, including the code generator and the target language, live in fact inside the same Gallina language is a strong asset for proof automation.

Verify low-level programs. The formal verification of the low-level programs involved here requires going beyond the features of existing frameworks for verified code generation, which operate on functional programs, generate high-level code, and may not handle pointers, nor I/O [58, 116]. Also, contrarily to the pure fragment of our high-level language, parallelism and especially concurrency no longer come for free. With the help of expert colleagues, we would build a verification framework based on the latest approach for separation logic proofs [114]. But an important point is to extend this approach so as to include the generation of code. Besides, our setting is significantly more subtle, as the programs we need to verify can manipulate arbitrary high-level objects, e.g., expressions. In such cases, separation logic proofs would nonetheless have to guarantee that the trusted code base remains safe.

Transport mathematical properties. As discussed in Chapter 2, paper proofs elide proof steps pertaining to folklore abstract algebra, thus the corresponding

proof steps in formal proofs have to be automated. The reconstruction is concerned with the needs of sharing for formal proofs (vocabulary and theories), for automation (delimited decision procedures), and for implementation (methods), which should all scale to the hierarchies underlying computer algebra systems. All existing approaches fail in at least one respect [89, 161, 78]. Proof search should also be informed with isomorphisms, e.g., between matrices of polynomials and polynomials of matrices. But as of today, no automation exist in formal systems to transport the properties of such isomorphic objects. Insights from homotopy theory [172] and parametricity for dependent types [163] provide a promising starting point, but how to turn these ideas into efficient concrete implementations remains to be invented.

5.3.3 Verify state-of-the art computational mathematics

Implementing the core of our computer algebra system, e.g., its arithmetic (typically a GMP-like library) and linear algebra routines (e.g., a subset of the BLAS library features), would be a benchmark for the outcome of the two first objectives. But I would also like to *use* this system and, crucially, to to make it usable enough for an audience of non-formal proof experts. Below are two applications that I think worth investigating, and a few comments on the improvements that need to be brought to a proof assistant like Coq to serve the needs of computer algebra users.

Special functions. What is the true meaning of the basic mathematical vocabulary such as *log* (the natural logarithm) or *Ai* (the Airy function)? The answer is clear on paper, much less so in silico. The specific formulas used in implementations are hardly exposed to users, next-to-impossible to reverse engineer. Hidden, incompatible assumptions are a major source of debilitating mistakes. By focusing on a restricted core of ∂-finite special functions [156], the ones mentioned in Chapter 3, one can aim at covering about 60% of the entries of the NIST Handbook of Mathematical Functions [139] in our environment, and address the needs of average users. My recent discussions with colleagues at the Vrije Universiteit Amsterdam also suggested a different nature of applications, which would be to provide a more specific algorithms toolbox for number theory, and more specifically for *modular forms*. Modular forms are a distinct class of special functions pervasive in mathematics and physics — and involved, e.g., in Wiles' proof of Fermat's last theorem, and in the Birch and Swinnerton-Dyer's conjecture [178] in number theory. An interesting case study would be the verification of the *classical modular forms data*, in the reference *L-functions and Modular Forms Database* [2].

User interface, inspection and visualization tools. Proof assistants today are not equipped with appropriate interaction tools for the needs of computer algebra users: for instance, available inspection tools are focused on logic rather than on user-library specific data-structures and notations. Providing a suite of fine-grained inspection tools, key to the usability of the environment, is a technical challenge, because of the need to address and connect every language layer. But this nonetheless represents a key ingredient to the usability of the software. It is all the more challenging that the feedback provided today to the users of proof assistants, and of its automation component is more than often hardly informative. At the formal proof level, the latter should provide facilities to uncover part of the elaboration process on any sub-term of a formula, improve error messages and cross-linking to documentation; at the domain-specific language level, it should provide tools for inspecting, visualizing, plotting mathematical objects, but also for querying their level of verification. The several technical skill necessary to devise such improvements are certainly outside of my current culture, but I would be keen on collaborating with the appropriate people to advance such topics.

5.4 Conclusion

I am convinced that it is possible to turn a proof assistant like Coq into a high-level, performance-oriented programming language, designed for writing efficient and correct code easily, and for serving the front-line of research in computational mathematics. In this environment, users would ideally be able to experiment with fast implementations, that do not need to be all formally verified at once. But users may also ultimately decide to increase the level of trust in any component at stake in their work. The same environment would therefore provide powerful verification tools, that allow to verify with high-productivity any component involved in a program, so as to prevent run-time errors, but also incorrect mathematical semantics.

Figure 5.2 gives an overview of what such an environment, based on Coq, could look like. Its purpose would be to help users producing and verifying computer-aided mathematics, expressed as theorems in CIC, the logical formalism underlying the Coq proof assistant. CIC is the top-most layer on the figure. The middle layer is Gallina, Coq's programming language, and the lowest layer is machine code. Different code generation phases, represented with dark arrows, refine an existential properties down to executable code. Light arrows represent code usage features; they allow the use of efficient, unverified code, for experimentation purposes, but when the corresponding code generation phases are formally verified, they also close the feedback loop to formal proofs.

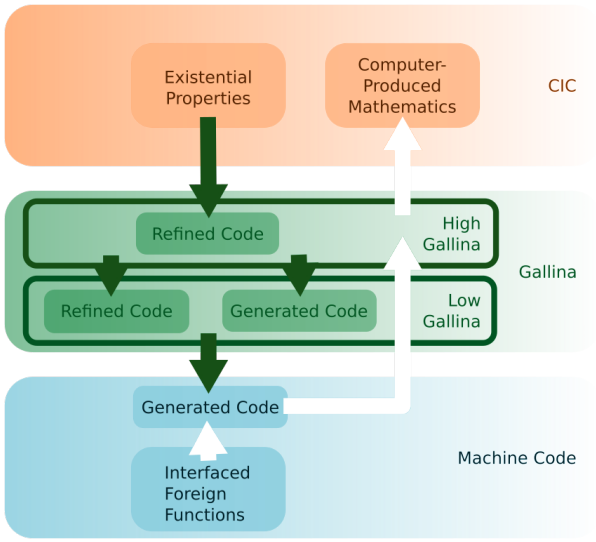

Figure 5.2: Architecture of the environment

Two decades ago, William Thurston wrote: "*formalizing parts of mathematics by computer, with actual formally correct formal deductions [...] is a very big but very worthwhile project, and I am confident that we will learn a lot from it. The process will help simplify and clarify mathematics*" [167]. But an appropriate environment for making this tangible still does not exist yet; this research program aims at contributing to its advent. I have the strong belief that formalized mathematics will bring even more to research in mathematics than just simplification and clarification; this material offers novel experimentation and learning perspectives, and it will ultimately allow for the discovery of new mathematics.

Bibliography

[1] FREAK: Factoring RSA export keys. https://mitls.org/pages/attacks/SMACK#freak, 2015.

[2] The L-functions and modular forms database. https://www.lmfdb.org, 2020.

[3] The lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020.

[4] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *3rd Annual Symposium on Logic in Computer Science (LICS)*, pages 165–175. IEEE Computer Society, July 1988.

[5] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[6] Sergei Ivanovich Adyan. Die Unentscheidbarkeit gewisser algorithmischer Probleme der Gruppentheorie. *Tr. Mosk. Mat. O.-va*, 6:231–298, 1957.

[7] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013.

[8] Algolib. http://algo.inria.fr/libraries/, 2013. Version 17.0. For Maple 17.

[9] Xavier Allamigeon, Ricardo D. Katz, and Pierre-Yves Strub. Formalizing the face lattice of polyhedra. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2020.

[10] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards. *IACR Cryptology ePrint Archive*, 2019:1155, 2019.

[11] Kenneth Appel and Wolfgang Haken. *Every planar map is four colorable*, volume 98 of *Contemporary Mathematics*. American Mathematical Society, Providence, RI, 1989. With the collaboration of J. Koch.

[12] Roger Apéry. Irrationalité de $\zeta(2)$ et $\zeta(3)$. *Astérisque*, 61, 1979. Société Mathématique de France.

[13] M. Aschbacher. *Finite Group Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2 edition, 2000.

[14] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[15] Jeremy Avigad. Type inference in mathematics. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, (106):78–98, 2012.

[16] Jeremy Avigad and John Harrison. Formally verified mathematics. *Commun. ACM*, 57(4):66–75, 2014.

[17] Clemens Ballarin. Locales: A module system for mathematical theories. *J. Autom. Reason.*, 52(2):123–153, 2014.

[18] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pak. The role of the mizar mathematical library for interactive proof development in mizar. *J. Autom. Reasoning*, 61(1-4):9–32, 2018.

[19] Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, September 2005.

[20] Bruno Barras. Sets in coq, coq in sets. *J. Formalized Reasoning*, 3(1):29–48, 2010.

[21] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *J. Funct. Program.*, 13(2):261–293, 2003.

[22] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33:494–566, 2020.

[23] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[24] Helmut Bender and Georges Glauberman. *Local analysis for the Odd Order Theorem*. Number 188 in London Mathematical Society, LNS. Cambridge University Press, 1994.

[25] Alexandre Benoit, Mioara Joldeş, and Marc Mezzarobba. Rigorous uniform approximation of D-finite functions using Chebyshev expansions. *Math. Comp.*, 86(305):1303–1341, 2017.

[26] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[27] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *J. Autom. Reason.*, 29(3-4):225–252, 2002.

[28] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 48–62, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[29] F. Beukers. A note on the irrationality of $\zeta(2)$ and $\zeta(3)$. *Bull. London Math. Soc.*, 11(3):268–272, 1979.

[30] Manjul Bhargava and John Hanke. Universal quadratic forms and the 290-theorem. *Inventiones mathematicae*. To appear.

[31] Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York-Toronto, Ont.-London, 1967.

[32] Jasmin Blanchette, Cezary Kaliszyk, Lawrence Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[33] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formal proof of the Lax–Milgram theorem. In *6th ACM SIGPLAN Conference on Certified Programs and Proofs*, Paris, France, January 2017.

[34] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, Mar 2015.

[35] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, and Éric Schost. Algorithmes efficaces en calcul formel, August 2017. 686 pages. Édition 1.0.

[36] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 182–194. ACM, 2017.

[37] Nicholas Bourbaki. The Architecture of Mathematics. *The American Mathematical Monthly*, 57(4), 1950.

[38] Nicolas Bourbaki. *General Topology*. Springer, 1995. Original French edition published by MASSON, Paris, 1971.

[39] Florent Bréhard, Nicolas Brisebarre, and Mioara Joldes. Validated and numerically efficient Chebyshev spectral methods for linear ordinary differential equations. *ACM Transactions on Mathematical Software*, 2018.

[40] Florent Bréhard, Assia Mahboubi, and Damien Pous. A certificate-based approach to formally verified approximations. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[41] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 299–312, January 2019.

[42] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Autom. Reason.*, 41(1):33–59, 2008.

[43] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 418–430. ACM, September 2011.

[44] Claude Chevalley and Henri Cartan. Schémas normaux ; morphismes ; ensembles constructibles. *Séminaire Henri Cartan*, 8, 1955-1956. talk:7.

[45] Laurent Chicli, Loic Pottier, and Carlos Simpson. Mathematical quotients and quotient types in coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2002.

[46] C. T. Chong and Y. K. Leong. An interview with Jean-Pierre Serre. *Math. Intelligencer*, 8(4):8–13, 1986.

[47] Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi. A computer-algebra-based formal proof of the irrationality of $\zeta(3)$. In Ruben Gamboa Gerwin Klein, editor, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.

[48] Frédéric Chyzak. *The ABC of Creative Telescoping: Algorithms, Bounds, Complexity*. Memoir of accreditation to supervise research (HDR), Université d'Orsay, April 2014. 64 pages.

[49] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *CPP*, volume 8307 of *LNCS*, pages 147–162. Springer International Publishing, 2013.

[50] Cyril Cohen and Assia Mahboubi. A formal quantifier elimination algorithm for algebraically closed fields. In *Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus 2010)*, volume 6167 of *Lecture Notes in Artificial Intelligence*, pages 189–203. Springer, 2010.

[51] Cyril Cohen and Assia Mahboubi. Formal proof in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Sciences*, 8(1:2):1–40, 2012.

[52] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in coq with elpi (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[53] Pierre Colmez and Jean-Pierre Serre, editors. *Grothendieck-Serre correspondence: Bilingual Edition.* American Mathematical Society - Société Mathematique de France, 2004.

[54] The Coq Development Team. *The Coq Proof Assistant, version 8.10.0*, October 2019.

[55] Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *European Conference on Computer Algebra (EUROCAL)*, volume 203 of *LNCS*, pages 151–184. Springer, April 1985.

[56] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *International Conference on Computer Logic (COLOG)*, volume 417 of *LNCS*, pages 50–66. Springer, December 1988.

[57] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.

[58] Nathanaël Courant, Antoine Séré, and Natarajan Shankar. The correctness of a code generator for a functional language. In Dirk Beyer and Damien Zufferey, editors, *21st International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 68–89, Cham, 2020. Springer International Publishing.

[59] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *26th International Conference on Automated Deduction (CADE)*, volume 10395 of *LNCS*, pages 220–236. Springer, August 2017.

[60] Paulo Emílio de Vilhena and Lawrence C. Paulson. Algebraically closed fields in isabelle/hol. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2020.

[61] David Delahaye and Micaela Mayero. Dealing with algebraic expressions over a field in Coq using Maple. *J. Symb. Comput.*, 39(5):569–592, 2005.

[62] Jose Divasón, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. A formalization of the LLL basis reduction algorithm. In Jeremy Avigad and Assia Mahboubi, editors, *9th International Conference on Interactive Theorem Proving (ITP)*, volume 10895 of *LNCS*, pages 160–177. Springer, July 2018.

[63] Boris Djalal. A constructive formalisation of semi-algebraic sets and functions. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 240–251. ACM, 2018.

[64] Christian Doczkal and Damien Pous. Graph theory in coq: Minors, treewidth, and isomorphisms. *J. Autom. Reason.*, 64(5):795–825, 2020.

[65] John W. Eaton, Sören Hauberg David Bateman, and Rik Wehbring. *GNU Octave version 3.8.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2014. ISBN 1441413006.

[66] Manuel Eberl. The irrationality of $\zeta(3)$. *Archive of Formal Proofs*, December 2019. http://isa-afp.org/entries/Zeta_3_Irrational.html, Formal proof development.

[67] Manuel Eberl. Nine chapters of analytic number theory in isabelle/hol. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 16:1–16:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[68] Manuel Eberl. Verified real asymptotics in Isabelle/HOL. In James H. Davenport, Dongming Wang, Manuel Kauers, and Russell J. Bradford, editors, *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 147–154. ACM, July 2019.

[69] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. In *22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 1 of *PACMPL*, pages 34:1–34::29, September 2017.

[70] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE, May 2019.

[71] Martín Hötzel Escardó. Synthetic topology: of data types and classical spaces. *Electron. Notes Theor. Comput. Sci.*, 87:21–156, 2004.

[72] Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3):775–1029, 1963.

[73] Bei-ye Feng. A simple elementary proof for the inequality $d_n < 3^n$. *Acta Math. Appl. Sin. Engl. Ser.*, 21(3):455–458, 2005.

[74] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/Hecke: Computer algebra and number theory packages for the Julia programming language. *CoRR*, abs/1705.06134, 2017.

[75] Stéphane Fischler. Irrationalité de valeurs de zêta (d'après Apéry, Rivoal, ...). In *Séminaire Bourbaki. Volume 2002/2003. Exposés 909–923*, pages 27–62, ex. Paris: Société Mathématique de France, 2004.

[76] Stéphane Fischler, Johannes Sprang, and Wadim Zudilin. Many odd zeta values are irrational. *Compositio Mathematica*, 155(5):938–952, 2019.

[77] Mathias Fleury, Jasmin Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In June Andronick and Amy P. Felty, editors, *7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 158–171. ACM, January 2018.

[78] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher-Order Logics (TPHOL2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[79] Georges Gonthier. Formal proof—the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008.

[80] Georges Gonthier. Point-free, set-free concrete linear algebra. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2011.

[81] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer Berlin Heidelberg, 2013.

[82] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[83] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher Order Logics (TPHOL2007)*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.

[84] Kiran Gopinathan and Ilya Sergey. Certifying certainty and uncertainty in approximate membership query structures. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2020.

[85] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 119–130. ACM Press, 1978.

[86] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 235–246. ACM, 2002.

[87] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005.

[88] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In Ulrich Furbach and Natarajan Shankar, editors, *3rd International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 423–437. Springer, August 2006.

[89] Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In Thorsten Altenkirch and Conor McBride, editors, *13th International Workshop on Types for Proofs and Programs (TYPES)*, volume 4502 of *LNCS*, pages 160–174. Springer, April 2006.

[90] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017.

[91] Denis Hanson. On the product of the primes. *Canad. Math. Bull.*, 15:33–37, 1972.

[92] John Harrison. Towards self-verification of HOL light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[93] John Harrison. Formal proofs of hypergeometric sums (dedicated to the memory of andrzej trybulec). *Journal of Automated Reasoning*, 55:223–243, 2015.

[94] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *J. Autom. Reason.*, 21(3):279–294, December 1998.

[95] Joel Hass and Roger Schlafly. Double bubbles minimize. *Ann. of Math. (2)*, 151(2):459–515, 2000.

[96] Harald Helfgott. *The ternary Goldbach problem.* Annals of Mathematical Studies, To appear.

[97] Harald A. Helfgott and David J. Platt. Numerical verification of the ternary Goldbach conjecture up to $8.875 \cdot 10^{30}$. *Experimental Mathematics*, 22(4):406–409, 2013.

[98] Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *32nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 6598–6606. AAAI Press, February 2018.

[99] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *LNCS*, pages 228–245. Springer, July 2016.

[100] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning (IJCAR)*, volume 10900 of *LNCS*, pages 646–662. Springer, July 2018.

[101] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *4th International Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 279–294. Springer, July 2013.

[102] Allan Hungria, Jean-Philippe Lessard, and Jason D. Mireles James. Rigorous numerics for analytic solutions of differential equations: the radii polynomial approach. *Math. Comp.*, 85(299):1427–1459, 2016.

[103] Yu. Ilyashenko. Centennial history of Hilbert's 16th problem. *Bull. Amer. Math. Soc. (N.S.)*, 39(3):301–354, 2002.

[104] Fabian Immler. A verified ODE solver and the Lorenz attractor. *J. Autom. Reasoning*, 61(1-4):73–111, 2018.

[105] Fredrik Johansson. Arb: a C library for ball arithmetic. *ACM Comm. Computer Algebra*, 47(3/4):166–169, 2013.

[106] Fredrik Johansson. Reliable real computing, October 2019. http://fredrikj.net/math/oxford2019.pdf.

[107] Mioara Joldes. *Rigorous Polynomial Approximations and Applications. (Approximations polynomiales rigoureuses et applications)*. PhD thesis, École normale supérieure de Lyon, France, 2011.

[108] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–259. ACM, January 2015.

[109] Cezary Kaliszyk and Freek Wiedijk. Certified computer algebra on top of an interactive theorem prover. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *14th Calculemus Symposium*, volume 4573 of *LNCS*, pages 94–105. Springer, June 2007.

[110] Edgar W Kaucher and Willard L Miranker. *Self-validating numerics for function space problems: Computation with guarantees for differential and integral equations*, volume 9. Elsevier, 1984.

[111] Manuel Kauers. Algorithms for D-finite functions. http://www.lifl.fr/jncf2015/files/lecture-notes/kauers.pdf. Journées Nationales de Calcul Formel 2015.

[112] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[113] Christoph Koutschan, Manuel Kauers, and Doron Zeilberger. Proof of George Andrews's and David Robbins's q-TSPP conjecture. *Proceedings of the National Academy of Sciences*, 108(6):2196–2199, January 2011.

[114] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. In *23rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 2 of *PACMPL*, pages 77:1–77:30, September 2018.

[115] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in coq. *Logical Methods in Computer Science*, 9(1), 2011.

[116] Peter Lammich. Refinement to imperative HOL. *J. Autom. Reasoning*, 62(4):481–503, 2019.

[117] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM Press, 2006.

[118] Jean-Philippe Lessard and Christian Reinhardt. Rigorous numerics for nonlinear differential equations using Chebyshev series. *SIAM J. Numer. Anal.*, 52(1):1–22, 2014.

[119] Robert Y. Lewis. An extensible ad hoc interface between Lean and Mathematica. In Catherine Dubois and Bruno Woltzenlogel Paleo, editors, *5th Workshop on Proof eXchange for Theorem Proving (PxTP)*, volume 262 of *EPTCS*, pages 23–37, September 2017.

[120] Assia Mahboubi. The rooster and the butterflies. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics*, volume 7961 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2013.

[121] Assia Mahboubi. Machine-checked mathematics. *Nieuw Archief voor Wiskunde*, 5/17(3):5, September 2016.

[122] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, volume 9807 of *Lecture Notes in Computer Science*. Springer, 2016.

[123] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *Journal of Automated Reasoning*, Mar 2018.

[124] Assia Mahboubi and Thomas Sibut-Pinote. A formal proof of the irrationality of $\zeta(3)$. *CoRR*, abs/1912.06611, 2019.

[125] Assia Mahboubi and Enrico Tassi. Canonical structures for the working Coq user. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *4th Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 19–34. Springer Berlin Heidelberg, July 2013.

[126] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, October 2020.

[127] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. In *24th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 3 of *PACMPL*, pages 104:1–104:29, August 2019.

[128] Kyoko Makino and Martin Berz. Verified computations using Taylor models and their applications. In Alessandro Abate and Sylvie Boldo, editors, *10th International Workshop on Numerical Software Verification (NSV)*, volume 10381 of *LNCS*, pages 3–13. Springer, July 2017.

[129] Benoit Mandelbrot. *The Fractal Geometry of Nature*. Freeman and Co., 1982.

[130] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *J. Autom. Reason.*, 57(3):187–217, 2016.

[131] Sean McLaughlin and John Harrison. A proof-producing decision procedure for real arithmetic. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 295–314. Springer, July 2005.

[132] Guillaume Melquiond and Raphaël Rieu-Helft. A why3 framework for reflection proofs and its application to gmp's algorithms. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 2018.

[133] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[134] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[135] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael

Norrish, editors, *3rd International Conference on Certified Programs and Proofs (CPP)*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81, Melbourne, Australia, December 2013.

[136] Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1994.

[137] Nedialko S. Nedialkov. Interval tools for odes and daes. In *12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006)*, pages 4–4, 2006.

[138] Russell O'Connor. Classical mathematics for a constructive world. *Math. Struct. Comput. Sci.*, 21(4):861–882, 2011.

[139] Frank W. J. Olver, Daniel W. Lozier, Ronald F. Boisvert, and Charles W. Clark. *NIST Handbook of mathematical functions*. Cambridge University Press, 2010.

[140] The PARI Group, Univ. Bordeaux. *PARI/GP version 2.11.0*, 2018.

[141] Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28, 2020.

[142] Thomas Peterfalvi. *Character Theory for the Odd Order Theorem*. Number 272 in London Mathematical Society, LNS. Cambridge University Press, 2000.

[143] Marko Petkovsek, Herbert S. Wilf, and Doron Zeilberger. $A = B$. A K Peters Series. Taylor & Francis, 1996.

[144] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. In *22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 1 of *PACMPL*, pages 17:1–17:29, September 2017.

[145] Michael O. Rabin. Recursive unsolvability of group theoretic problems. *Annals of Mathematics*, 67(1):172–194, 1958.

[146] René Thom. *Prédire n'est pas expliquer*. Champs. Flammarion, 2009.

[147] Egbert Rijke. *Classifying Types*. PhD thesis, Department of Philosophy, Carnegie Mellon University, 2018.

[148] Robert H. Risch. The solution of the problem of integration in finite terms. *Bull. Amer. Math. Soc.*, 76(3):605–608, 05 1970.

[149] Tanguy Rivoal. *Propriétés diophantiennes de la fonction zêta de Riemann aux entiers impairs*. PhD thesis, Université de Caen, 2001.

[150] Damien Rouhling. A formal proof in coq of a control function for the inverted pendulum. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 28–41. ACM, 2018.

[151] Damien Rouhling. *Outils pour la Formalisation en Analyse Classique – Une Étude de Cas en Théorie du Contrôle*. PhD thesis, Université Côte d'Azur, 2019.

[152] Siegfried M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. http://www.ti3.tuhh.de/rump/.

[153] Siegfried M. Rump. Function "abs" in INTLAB. INTLAB users mailing list, May 2016.

[154] Kazuhiko Sakaguchi. Validating mathematical structures. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2020.

[155] Bruno Salvy. An Algolib-aided version of Apéry's proof of the irrationality of $\zeta(3)$. http://algo.inria.fr/libraries/autocomb/Apery2-html/apery.html, 2003.

[156] Bruno Salvy. Linear differential equations as a data-structure. *Foundations of Computational Mathematics*, 19(5):1071–1112, 2019.

[157] Romain Serra, Denis Arzelier, Mioara Joldes, Jean-Bernard Lasserre, Aude Rondepierre, and Bruno Salvy. Fast and accurate computation of orbital collision probability for short-term encounters. *Journal of Guidance, Control, and Dynamics*, 39(5):1009–1021, 2016.

[158] Ronald Solomon. A brief history of the classification of the finite simple groups. *Bull. Amer. Math. Soc. (N.S.)*, 38(3):315–352, 2001.

[159] Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with Taylor interval approximations. *CoRR*, abs/1301.1702, 2013.

[160] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[161] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011.

[162] Mark A. Stadtherr. High performance computing: Are we just getting wrong answers faster? *AIChE CAST Communications*, 22(1):6–14, 1999.

[163] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *PACMPL*, 2(ICFP):92:1–92:29, 2018.

[164] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[165] Alfred Tarski. A decision method for elementary algebra and geometry. Technical report, RAND Corporation, Santa Monica CA, 1948. Republished as A Decision Method for Elementary Algebra and Geometry, 2nd ed. Berkeley, CA: Universityof California Press, 1951.

[166] John Tate. Correspondance grothendieck-serre. *Nieuw Archief vor Wiskunde*, March 2004.

[167] William P. Thurston. On proof and progress in mathematics. *Bull. Amer. Math. Soc. (N.S.)*, 30(2):161–177, 1994.

[168] Andrzej Trybulec and Howard A. Blair. Computer assisted reasoning with MIZAR. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*, pages 26–28. Morgan Kaufmann, 1985.

[169] Warwick Tucker. A rigorous ODE solver and Smale's 14th problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.

[170] Warwick Tucker. *Validated numerics*. Princeton University Press, Princeton, NJ, 2011. A short introduction to rigorous computations.

[171] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming. *CoRR*, abs/1908.05647, 2019.

[172] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[173] Jan Bouwe Van Den Berg and Jean-Philippe Lessard. Chaotic braided solutions via rigorous numerics: Chaos in the Swift–Hohenberg equation. *SIAM Journal on Applied Dynamical Systems*, 7(3):988–1031, 2008.

[174] Alfred van der Poorten. A proof that Euler missed: Apéry's proof of the irrationality of $\zeta(3)$. *Math. Intelligencer*, 1(4):195–203, 1979. An informal report.

[175] Cédric Villani. *Birth of a Theorem: A Mathematical Adventure*. Penguin Books, 2016.

[176] Markus Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

[177] Freek Wiedijk. Pollack-inconsistency. *Electron. Notes Theor. Comput. Sci.*, 285:85–100, 2012.

[178] Andrew Wiles. The Birch and Swinnerton-Dyer conjecture, 2016. Clay Mathematics Institute.

[179] Wolfram Language & System Documentation Center. *Symbolic Parameters and Inexact Quantities*, November 2019. https://reference.wolfram.com/language/tutorial/DSolveSymbolicAndInexactQuantities.html.

[180] Wolfram Language & System Documentation Center. *Symbolic Parameters and Inexact Quantities*, November 2019. https://reference.wolfram.com/language/guide/FunctionalProgramming.html.

[181] Nobito Yamamoto. A numerical verification method for solutions of boundary value problems with local uniqueness by Banach's fixed-point theorem. *SIAM J. Numer. Anal.*, 35(5):2004–2013, 1998.

[182] Doron Zeilberger. Closed form (pun intended!). In *A tribute to Emil Grosswald: number theory and related analysis*, pages 579–607. Providence, RI: American Mathematical Society, 1993.

[183] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536, 2017. https://eprint.iacr.org/2017/536.

[184] Wadim V. Zudilin. One of the numbers $\zeta(5)$, $\zeta(7)$, $\zeta(9)$, $\zeta(11)$ is irrational. *Uspekhi Mat. Nauk*, 56(4(340)):149–150, 2001.

Titre : Mathématiques Assistées et Vérifiées par Ordinateur

Mot clés : Théorie des types, preuves formelles, calcul formel

Résumé : Les assistants de preuve sont des logiciels conçus pour la réalisation de bibliothèques de mathématiques numérisées. Celles-ci contiennent des définitions, énoncés et preuves tous formalisés dans une variante de logique fixée, de sorte que la vérification de la bonne formation des énoncés, et la correction des preuves, puissent être réduites à un processus mécanique, celui associé au formalisme logique sous-jacent. Le noyau de l'assistant de preuve est le composant du logiciel qui réalise cette vérification, tandis que l'assistant de preuve à proprement parler implémente un ensemble de techniques d'automatisation qui permettent à ses utilisateurs de mener à bien la formalisation en pratique de définitions et de théories mathématiques arbitrairement sophistiquées.

Ce mémoire présente une synthèse de trois contributions principales à la vérification formelle de théories mathématiques en théorie des types dépendants. La première de ces contributions porte sur la réalisation d'une bibliothèque de mathématiques formalisées couvrant les résultats classiques d'algèbre de niveau licence, ainsi que des pans plus avancés de théorie des groupes finis. Les deux autres contributions concernent les enjeux de vérification formelle de preuves mathématiques calculatoires, respectivement au moyen d'algorithmes symboliques et de méthodes numériques rigoureuses.

Title: Machine-Checked Computer-Aided Mathematics

Keywords: Type theory, formal proofs, computer algebra

Abstract: Proof assistants are pieces of software designed for the realization of digital libraries of formalized mathematics. The latter libraries contain definitions, statements, and proofs, all formalized in a fixed variant of logic. In particular, the verification of the well-formedness of statements, and of the correctness of proofs, boils down to a mechanical process, associated with the underlying logical formalism. The kernel of a proof assistant is the software component which performs this verification, while the actual proof assistant implements a collection of automation techniques, which allow users to conduct in practice the formalization of arbitrarily sophisticated mathematical definitions and theories.

This memoir presents an overview of three main contributions to the formal verification of mathematical theories in dependent type theory. The first of these contributions deals with the realization of a library of digitized mathematics covering the standard undergraduate background in algebra, as well as some more advanced chapters in finite group theory. The two other contributions are related to the issues pertaining to the formal verification of computational mathematical proofs, by the means of symbolic algorithms and of rigorous numerical methods respectively.