Investigations in Computer-Aided Mathematics : Experimentation, Computation, and Certification

Thomas Sibut Pinote

▶ To cite this version:

Thomas Sibut Pinote. Investigations in Computer-Aided Mathematics : Experimentation, Computation, and Certification. Computation and Language [cs.CL]. Université Paris Saclay (COmUE), 2017. English. ⟨NNT : 2017SACLX086⟩. ⟨tel-01691185⟩

HAL Id: tel-01691185

https://pastel.hal.science/tel-01691185v1

Submitted on 23 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

université PARIS-SACLAY

ÉCOLE POLYTECHNIQUE UNIVERSITÉ PARIS-SACLAY

NNT : 2017SACLX086

Thèse de doctorat de l'Université Paris-Saclay préparée à l'École Polytechnique

Ecole doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique

par

M. Thomas Sibut-Pinote

Investigations en Mathématiques Assistées par Ordinateur: Expérimentation, Calcul et Certification

Composition du Jury :

Mme	Sylvie Boldo	Directrice de Recherche Inria Saclay, Université Paris-Saclay	(Présidente du Jury)
M.	Bruno Salvy	Directeur de Recherche Inria, ENS de Lyon	(Rapporteur)
M.	John Harrison	Ingénieur de Recherche Intel	(Rapporteur)
Mme.	Assia Mahboubi	Chargée de Recherche Inria	(Directrice de thèse)
M.	Paul Zimmermann	Directeur de Recherche Inria Nancy, Loria	(Examinateur)
M.	Pierre-Yves Strub	Maître de Conférences École Polytechnique	(Examinateur)
M.	Laurent Théry	Chargé de Recherche Inria Sophia Antipolis	(Examinateur)
M.	Lawrence Paulson	Professeur University of Cambridge	(Examinateur)

Remerciements

Tout d'abord, je veux remercier ma directrice de thèse Assia Mahboubi pour sa confiance, sa pédagogie et sa patience infinie.

Je veux également remercier ceux avec qui j'ai travaillé directement: Frédéric Chyzak, Guillaume Melquiond, Éric Schost, Pierre-Yves Strub, Georges Gonthier et Santiago Zanella-Beguelin.

Je remercie les rapporteurs de ma thèse, Bruno Salvy et John Harrison, pour leur relecture de mon manuscrit ainsi que Sylvie Boldo, Laurent Théry, Pierre-Yves Strub, Lawrence Paulson et Paul Zimmermann pour avoir accepté de faire partie de mon jury de thèse.

Je remercie toute l'équipe Specfun, ceux qui y ont passé ou passent encore du temps, et ceux qui sont dans ses parages: Alin, Christine, Cyril, Damien, Hélène, Philippe, Pierre, Laurent, Louis, Marc, Rémi, Rémy, Suzy, qu'il s'agisse de discussions scientifiques ou de bons moments passés ensemble autour de la machine à café ou d'un repas.

Je voudrais remercier Denis Choimet pour son dévouement à faire aimer les mathématiques à ses élèves, et pour m'avoir donné confiance en moi.

Je remercie également tous les chercheurs avec qui j'ai échangé avant et/ou pendant ma thèse et qui m'ont aidé à m'orienter: j'en oublie sûrement et je les prie par avance de m'en excuser, mais je pense notamment à Daniel Hirschkoff, Guillaume Hanrot, Nicolas Brisebarre, Mathieu Sozeau, et Daniel Augot.

Je voudrais aussi remercier les institutions qui ont rendu possibles ma thèse et mes stages: l'ENS de Lyon, l'École Polytechnique, l'Inria et Microsoft Research.

Je salue Zach Weiner, auteur de "SMBC comics", ainsi que Satoshi Nakamoto, génial inventeur de Bitcoin: j'ai consacré à leurs oeuvres respectives une part parfois peu raisonnable de mon temps ces trois dernières années.

Merci à tous ceux qui ont fait le chemin de la thèse en même temps que moi pour l'avoir rendue plus agréable par leur compagnie, je pense en particulier à Sonia, Salomé, Loïc et Alice.

Je tiens à remercier tous ceux qui rentrent moins facilement dans une case mais qui tiennent une place dans mon coeur depuis des années, dans le désordre chronologique: Meriem, Antoine P., Gabriela, Floriane, Sarah, Natalia, Fosca, Juliette, Mathieu, Fanny, Antoine A. (aka Doudou), Florent C., Lucca, Marie-Charlotte, Jean-Florent. Il manque ceux que je connais depuis moins longtemps, ou bien que j'ai un peu perdu de vue ces derniers temps, et que je veux également remercier, en priant encore ceux que j'oublie de m'en excuser: Déborah, Katia, Adrien, Laure, Jessica, Danuta, Alix, Marion, Ava, Jonathan, Jasmina, Lauranne, Carmen, Olya, Florent et Thibaut D., Paul-Elliot, Julie M., Sarah M, Héloïse.

J'ai une pensée pour Tomas Tangarife, que j'ai peu connu mais dont le départ m'a beaucoup touché, ainsi que pour ses proches.

Merci à Hugo et Vincent d'être des frères formidables, et sur qui on peut toujours compter.

Je voudrais remercier du fond du coeur ma mère pour avoir toujours été là pour moi. Les moments difficiles nous ont rapprochés, et c'est un grand bonheur pour moi.

Je remercie mon père, Gaëtane, Elliot et Timothée qui m'ont toujours accueilli chaleureusement.

Merci à toute la famille, Mamie Suze, Papi Jean, Jean-Marc, Mamie Yop, Anne, Ben, Lola, Antoine, Thibaud, Alexandre, Aurélien, Jérémy, et leurs familles; j'ai également une pensée pour Papi Raymond.

Enfin, je voudrais remercier Laetitia à qui je dois tant, et qui fait de moi une meilleure personne.

Contents

1 Introduction **6**
- 1.1 Context . . . 6
- 1.2 Contributions . . . 9
  - 1.2.1 Experiment . . . 9
  - 1.2.2 Certifying Computations . . . 10
  - 1.2.3 Certified Computations . . . 10
- 1.3 Notations and Conventions . . . 10
- 1.4 Programming Languages and Libraries . . . 11
  - 1.4.1 Ocaml . . . 11
  - 1.4.2 Coq . . . 12

I Experiment **14**

2 Introduction **15**

3 Experimenting with a Tiling Conjecture **17**
- 3.1 Problem Description . . . 17
- 3.2 Exact Matrix Cover, The Dancing Links Algorithm and Combine 18
- 3.3 Solutions for $n = 6$ and $n = 7$ . . . 19
- 3.4 Conclusion . . . 21

4 Experiments with Matrix Multiplication Algorithms **22**
- 4.1 Definitions . . . 22
  - 4.1.1 Matrix Multiplication . . . 22
  - 4.1.2 Rank of a bilinear map . . . 23
  - 4.1.3 Usual properties of the rank . . . 26
  - 4.1.4 Degeneration and border rank . . . 27
- 4.2 A Quick Overview of Some Matrix Product Algorithms . . . 29
  - 4.2.1 Strassen's algorithm (1969) . . . 29
  - 4.2.2 Bini's Approximate Algorithms (1979) . . . 29
  - 4.2.3 Schönhage's τ-theorem (1981) . . . 30
  - 4.2.4 Pan's Trilinear aggregation (1984) . . . 30
- 4.3 A Constructive Proof of the τ-Theorem and a New Family of Algorithms . . . 32
  - 4.3.1 Proof . . . 32
  - 4.3.2 Removing ε's and counting operations . . . 35
- 4.4 Scaling the Experiment Up: Implementations from Maple to Ocaml 38

4.4.1 Maple Implementation: Description, Advantages and Downsides . . . 38
4.4.2 Ocaml Implementation . . . 41
4.4.3 Exact Complexity of Some Matrix Product Algorithms . 44
4.5 Conclusion . . . 46

II Certifying Computations 47

5 Introduction 48

6 History and outline of Apéry's theorem 51

7 Preliminaries 54
7.1 Integers . . . 54
7.2 Sharing theories and notations . . . 56
7.3 Algebraic numbers, real numbers . . . 57
7.4 Computations . . . 58

8 Formalizing Apéry's theorem in Coq 60
8.1 Recurrence and Creative Telescoping . . . 60
8.1.1 Recurrences as a data structure for sequences . . . 61
8.1.2 Apéry's sequences are ∂-finite constructions . . . 63
8.1.3 Provisos and sound creative telescoping . . . 63
8.1.4 Generated operators, hand-written provisos, and formal proofs . . . 65
8.1.5 Definitions of conditional recurrence predicates . . . 66
8.1.6 Formal proofs of a conditional recurrence . . . 67
8.1.7 Composing closures and reducing the order of B . . . 69
8.2 Consequences of Apéry's recurrence . . . 69
8.3 Asymptotics of $lcm(1, ..., n)$ . . . 71
8.3.1 Notations and Outline . . . 72
8.3.2 Proof . . . 72

9 Conclusion 77
9.1 Proving Inequalities in Coq . . . 77
9.2 Casts . . . 78
9.3 Related Work . . . 78

III Certified Computations 81

10 Introduction 82

11 Efficient and Rigorous Numeric Real Computations 85
11.1 Interval Arithmetic . . . 85
11.1.1 Notations and definitions . . . 85
11.1.2 Interval Arithmetic . . . 86
11.1.3 Interval Arithmetic and CoqInterval . . . 87
11.1.4 Representation of functions, efficient computations: straight-line programs . . . 88

11.2 Rigorous Polynomial Approximations 90
11.2.1 CoqApprox: Rigorous Polynomial Approximations and Taylor Models in Coq 90
11.2.2 Primitives of RPAs 91

12 Computing Estimates of Integrals in Coq 93
12.1 Estimating Proper Integrals 93
12.1.1 A Naive Approach 93
12.1.2 Another Approach using RPAs 94
12.1.3 Polynomial Approximations Are Not Always Better . . . 95
12.1.4 Dichotomy 96
12.1.5 Accuracy 96
12.1.6 Automatic Continuity and Integrability 98
12.2 Estimating Improper Integrals 99
12.2.1 Improper integral of a product 100
12.2.2 Catalog of supported integrable functions 101
12.2.3 Case of 0+ 102
12.2.4 Integrability 102
12.2.5 Extending Coquelicot 103
12.2.6 Conclusion 109
12.3 Benchmarks 110
12.3.1 Proper integrals 110
12.3.2 Improper integrals 112

13 Quadrature and Automatic Differentiation of order n 115
13.1 Quadrature Methods: A quick introduction 115
13.1.1 AD of order n 116
13.1.2 Test of Simpson's Method in Coq 122

14 Conclusion and Perspectives 124

IV Conclusion 127

A Complexity of Some Matrix Multiplication Algorithms 141

Chapter 1

Introduction

1.1 Context

Computers and computer science have progressively made themselves indispensable to mathematicians. This is of course true of communication with the ubiquitous use of search engines, of LaTeX [80] to write papers, of Math Overflow [115] to ask and answer questions from peers, of blogs [56] [105] to discuss and produce new results.

More directly, the vast computational power available today has allowed mathematicians to extend their ability to do mental computations: they can experiment further than before, try to find more complicated counter-examples to conjectures, visualize patterns on more terms of a sequence. In *An Introduction to Mathematics* [125], Whitehead claims about mathematical symbolism that "civilization advances by extending the number of important operations which we can perform without thinking about them". Specialized software (for numerical computations, computer algebra, etc...) have arguably participated in this extension.

Not only have computations at the service of mathematicians become more ambitious, some recent developments have even attempted to cut out the middleman in the mathematician's main product: proofs. The Boolean Satisfiability Problem (SAT) consists in determining, given a formula ϕ involving boolean variables $x_1, \cdots, x_n$ and the *and* ($\wedge$) and *or* ($\vee$) connectives, whether there exist boolean values $b_1, \cdots, b_n$ such that when substituting b_i for x_i, ϕ evaluates to *true*. For example, $x_1 \wedge (\neg x_2 \vee \neg x_1)$ is satisfiable with the assignment $x_1 := \textit{true}$ and $x_2 := \textit{false}$. Thus, a successful assignment of boolean values to the variables of ϕ constitutes, up to a routine verification which can be accomplished by a simple program, a *proof* of the satisfiability of ϕ. Powerful solvers for the SAT problem (called SAT-solvers) have been developed. Surprisingly, complex problems can be reduced to the satisfiability of a (potentially large) boolean formula. The Pythagorean Triples Problem asks whether for all partitions of the set $\{1, 2, \cdots, n\}$ into 2 parts, at least one partition contains a Pythagorean triple. A recent paper [72] shows that it holds using a SAT solver, producing a 200-terabyte file. Finding proofs based on this method raises at least two questions. A first one is that of whether we can trust the proof checker as a program. This is a small instance of an issue which will be addressed in the next paragraph.

Secondly, the boolean encoding changes the statement of the problem so that if a proof is found, it does not follow the structure of the original problem. Thus, the proof may carry little to no insight as to *why* the theorem is true. About an unintelligible 1500 page computer-generated computer proof [103], Hales argues that "eventually, we will have to content ourselves with fables that approximate the content of a computer proof in terms that humans can comprehend" [61].

If we stay on the side of proofs made and checked by humans, many mathematical proofs in the past century have come to *rely* decisively on computations; we list several examples from various domains of mathematics. The Four Color Theorem, stating that every planar graph can be colored with at most four colors so that no two adjacent nodes bear the same color, was first proved in 1977 [4] by Appel and Haken by examining $1,936$ possible configurations using a computer. The Double Bubble conjecture, characterizing the shape which encloses two equal volumes with minimal surface as being made of two pieces of spheres separated by a flat disk, was first proved in 1995 by Hass, Hutchings and Schlafly. They reduced the proof to showing thousands of inequalities on simple integrals and wrote a custom `C++` program doing interval arithmetic which ran in 10 seconds at the time. Tucker solved Smale's 14th problem in 1999, proving that the Lorenz attractor is strange [120], notably by writing and using a program computing solutions to ordinary differential equations. Hales and Ferguson solved the Kepler conjecture in 1998 [60], establishing the densest way to pack spheres in three dimensions, originally needing 2000 hours of computations to solve huge nonlinear optimization problems. After four years, the 12 referees gave up and decided they could not possibly review the whole proof and computer code. In 2013, Helfgott proved the Ternary Goldbach Conjecture [71]. It states that every odd number n greater than 5 is the sum of three primes. The proof relies on two different natures of computations: the "manual" verification that the conjecture holds for $n \leq 10^{27}$, and the approximation of some numerical integrals. In order to compute the latter, he posted a question on Math Overflow [69] to ask for a tool doing rigorous integration.

Such proofs relying on heavy computations are controversial: programs tend to have bugs, and the number of bugs is reputed to be proportional to the number of lines of code. One single bug could render the whole proof invalid. For this reason, efforts have been made to dramatically increase confidence in these proofs. One possible route has been to make a new, "computer-free" proof of the same result, as a later paper [75] on the Double Bubble conjecture boasts. Another way has been to make computations as rigorous as possible, like Tucker who used interval arithmetic systematically in his proof. When the last two solutions are not suitable, there remains an ultimate step: building a complete mechanized formal proof of the result from elementary mathematical axioms to remove all possible doubt. This is the way chosen by Gonthier [50] with the help of Werner when they proved the Four Color Theorem in the Coq proof assistant in 2005 and by Hales [59] who formalized with his team his proof of the Kepler conjecture in the Isabelle and HOL Light proof assistants.

These proof assistants are pieces of software which allow to build proofs from first principles, i.e. logical axioms and inference rules. Despite Bourbaki's opinion that "formalized mathematics cannot in practice be written down in full", the increase in computational power has in fact made this practical and numerous theorem provers have flourished in the past three to four decades.

It is possible to adopt a *skeptical* approach to certifying computations in a

proof assistant. Suppose for example that we want to prove that some large number n is not prime. We could have a program produce two numbers $1 < p < n$ and q such that

$$n = p \cdot q. \tag{1.1}$$

Once we have the result, we can forget about the program and simply build a proof of Equation (1.1) inside a proof assistant. Here, the representation of natural numbers, the multiplication operation on them, and equality are underlying objects of the logic of this proof assistant.

The internal logic of the proof assistant can however also be made efficient enough to directly deal with non-trivial computations. In this case, the program doing the computation is written in a language internal to the logic and its result is natively accepted as valid. This is called an *autarkic* approach and it is the path chosen by Gonthier and Werner.

Whether using an autarkic or a skeptical approach to certifying computations, there still remains to formalize the remaining parts of a proof, which do not necessarily rely on computations. Furthermore, formalizations of mathematical proofs have not been limited to controversial, computation-heavy results; they have also been used for more traditional (but highly non-trivial) results, such as the Odd Order Theorem [52]. Though this was not a contested result, one otherwise strong motivation for doing formal proofs in general is to reduce the risk of human mistakes while writing proofs. This aspect convinced Fields medalist Voevodsky to turn to formal proofs after a false result which he published in 1991 [78] remained unquestioned until 1998, when Carlos Simpson showed that the result could not be true [114] [122]. Voevodsky finally found the mistake in his paper in 2013 [124].

Whether large proofs are computation-heavy or not, mathematical results from all the theories involved in the proof must be systematically formalized in order to mechanize them in proof assistants. Much like when programmers write complex software, this involves careful choices of data structures and statements. Tools for helping the users of proof assistants in their tasks must be developed. Empirical experience shows that what may seem like a "good enough" design choice at a small scale can become unbearable technical debt for larger projects.

In fact, whereas one might think that formalizing results consists in copying paper proofs in a slightly more rigorous way, this activity leads to fresh insights on sometimes well-established mathematical theories. The formalization of the Odd Order theorem led Gonthier to come up with an original way to model abstract linear algebra using only matrices as a data structure to represent spaces, families of vectors [51]. In his paper about the formalization of the Prime Number Theorem, Avigad [6] gives several examples of "dumbing down" proofs, that is of coming up with more elementary proofs of a result in the absence of a library providing the necessary theory to directly adapt a textbook proof.

Building tools to help users develop formal proofs is not only about building libraries of theorems, it is also about automating away part of the tedious work of proving every little detail in a proof. Having a routine which can automatically prove that two rational fractions are equal, or establish simple inequalities between linear expressions can greatly improve the experience of the user. This is another way of "extending the number of operations which can be performed without thinking about them".

In this thesis, we illustrate three different ways to use computers to build mathematical proofs. We describe how *experiments* with existing matrix multiplication algorithms led to the discovery of a new family of parallel matrix multiplication algorithms. We then turn to formal proofs. We fully formalize a theorem by Apéry, in big part by *certifying* the outputs of a Computer Algebra System. Finally, we build an extension of the CoqInterval library which can *compute* fully verified approximations of integrals in a *certified* way.

1.2 Contributions

In this section, we list the contributions made in this thesis, by order of appearance and grouped following the structure of this document. Whenever our work has been published or refers to code stored online, we include the appropriate references.

1.2.1 Experiment

The contents of this chapter are partly the result of unpublished work done during the author's Master's degree in an internship supervised by Éric Schost. Part of this work was presented at the Journées Nationales du Calcul Formel[1] *in 2015 in Cluny and continued during a three-week visit to University of Waterloo in November 2016. The theorems of Section 4.3 and the Maple code were done as a collaboration with Éric Schost. The Ocaml code is mostly our own work, though it benefited along the way from insights and contributions from Éric Schost. The whole Specfun team is to be thanked for helping us come up with a hopefully clearer presentation of these results.*

A New Class of Parallel Matrix Multiplication Algorithms We discover a new class of parallel matrix multiplication algorithms (Theorem 3). These algorithms can be obtained by applying a practical, finite version of Schönhage's τ-theorem to existing "approximation" algorithms by Pan and Schönhage and observing that one can remove the additional ε variable which had been introduced in these algorithms. We also provide a constructive proof of the τ-theorem which is inspired from this result.

A Software Tool for Manipulating Matrix Multiplication Algorithms We provide a software tool for the symbolic manipulation of tensors encoding matrix multiplication algorithms. Our software can parse such tensors, compose them, apply the τ-theorem following our contribution in the previous paragraph, and export them to various formats including Maple, LaTeX and `C++`. This software requires `git`, and a recent version of Ocaml which can be installed using `opam`. It can be obtained and installed on Linux or Mac[2] by typing the following three commands:

```
git clone -b public https://gitlab.com/matrix-product-experiments/mpe
cd mpe/caml
sh build.sh
```

[1] Symbolic Computation National Days, a yearly meeting of the French community of Computer Algebra and Symbolic Computation.

[2] We have not tested it on a Mac, but there shouldn't be any specific counter-indication.

The specific options for using our tool are described in Section 4.4.2.

1.2.2 Certifying Computations

A Full Proof of Apéry's Theorem in Coq As a result of a collaboration with Frédéric Chyzak, Assia Mahboubi and Enrico Tassi, we provide a complete formal proof of Apéry's theorem stating the irrationality of the constant $\zeta(3)$ (Theorem 5). Part of this work has been published at ITP 2014 [26] and a longer version, which also includes a simplification and formalization of a proof by Hanson regarding the asymptotic behavior of the sequence $lcm(1,\cdots,n)$, is currently in the works. Our personal contribution is the development of a library of arithmetic and number theory results concerning the sequences of Apéry's proof, the simplification and formalization of Hanson's proof, and assisting Frédéric Chyzak and Assia Mahboubi in coming up with a paradigm to state sound creative telescoping statements, to develop tactics both to handle big integers and to automatically resolve some linear inequalities over $\mathbb{Z}$. Our Coq development can be found at `https://specfun.inria.fr/~tsibutpi/code/apery.zip`.

Questioning the status of outputs of creative telescoping algorithms as straightforward proofs The main part of our proof of Theorem 5 is a *validation* of the outputs of creative telescoping algorithms which were intended to be used as a *certificate*. We question their status as certificates providing complete proofs of identities (Section 8.1).

1.2.3 Certified Computations

This work was done in close collaboration with Assia Mahboubi and Guillaume Melquiond.

An extension of CoqInterval with proper and improper integrals We provide an extension of the CoqInterval library in order to approximate both improper and proper integrals using a mix of floating-point computations, interval arithmetic and rigorous polynomial approximations. All computations happen *inside* the logic of the Coq proof assistant. Using this tool, we highlight several irregularities in the published literature concerning integral computations. It has been published at ITP 2016 concerning proper integrals [86], and a longer article has been accepted and will appear in a special edition of the Journal of Automated Reasoning to be published in 2017 [87].

An extension of Coquelicot with a generalized Riemann integral In order to build the tool described in the previous paragraph, we extended the Coquelicot [82] [20] library so as to improve the handling of improper integrals (Section 12.2.5).

1.3 Notations and Conventions

In this section, we present some general notations. More context-specific notations will be introduced in each of the independent three parts.

In the context of both Ocaml and Coq code, as is usual, we will write function application in what is called a *currified* manner, that is $f\ x$ instead of $f(x)$. This generalizes to more parameters. For example, if f is a function with three parameters x, y and z, then we will write the application of f to x, y and z as $f\ x\ y\ z$. The partial application $f\ x\ y$ is a function which maps z to $f\ x\ y\ z$.

If $k \in \mathbb{N}$ and if $D \subseteq \mathbb{R}$, we will denote by $\mathcal{C}^k(D)$ the set of functions which are k times continuously differentiable on D, and by $\mathcal{C}^k$ the set $\mathcal{C}^k(\mathbb{R})$.

We will denote by $\mathbb{R}[X]$ the set of polynomials with real coefficients, and if $n \in \mathbb{N}$, $\mathbb{R}_n[X]$ will be the set of polynomials $P \in \mathbb{R}[X]$ of degree at most n.

If X is a finite set, we denote by $|X|$ the cardinal of X.

For $n, k_1, \cdots, k_l \in \mathbb{N}$, with $k_1 + \cdots + k_l = n$, the quantity $\binom{n}{k_1, \cdots, k_l} := \frac{n!}{k_1! \cdots k_l!}$ is called a *multinomial coefficient*. If $\mu \in \mathbb{N}^l$, we will write $\binom{n}{\mu}$ for $\binom{n}{\mu_1, \cdots, \mu_l}$. We also have

$$\binom{n}{k_1, \cdots, k_l} = \prod_{i=1}^{l} \binom{k_1 + \cdots + k_i}{k_i}. \tag{1.2}$$

In particular, $\binom{n}{k_1, \cdots, k_l} \in \mathbb{N}$. The multinomial theorem states that $\binom{n}{k_1, \cdots, k_l}$ is the coefficient of $x_1^{k_1} \cdots x_l^{k_l}$ in the formal expansion of $(x_1 + \cdots + x_l)^n$:

Theorem 1 (Multinomial Theorem)**.** *Let $\mathcal{A}$ be a commutative ring and let $x_1, \cdots, x_l \in \mathcal{A}$. Then for $n \in \mathbb{N}$,*

$$(x_1 + \cdots + x_l)^n = \sum_{k_1 + \cdots + k_l = n} \binom{n}{k_1, \cdots, k_l} x_1^{k_1} \cdots x_l^{k_l}.$$

where we take the convention $0^0 = 1$.

1.4 Programming Languages and Libraries

In this part, we present the various software tools used in the rest of the thesis.

First, Ocaml and Coq are two pieces of software from Inria whose histories are closely bound together, and which were used extensively in the works presented in this thesis. The functional programming language Ocaml was instrumental in Part I in order to build our software tool to manipulate tensors representing matrix product algorithms and in Part III to build prototypes which were later implemented in Coq. The Coq theorem prover was used to prove Apéry's theorem in Part II and to build and prove correct our extension of CoqInterval to approximate integrals in Part III.

1.4.1 Ocaml

Ocaml [83] is a general purpose programming language of the family of ML languages. ML is the meta-language of Milner's proof assistant LCF (Logic for Computable Functions) [94] [54]. One core idea of LCF was to represent provable theorems as elements of an abstract type `theorem` which could only be instantiated with predefined axioms and elementary inference rules. This way, any element of the type theorem would be provable *by construction*, since it had to have been built from first principles.

On top of predefined base types such as `nat` for natural numbers, `bool` for booleans and `float` for floating-point numbers, Ocaml provides the possibility to define *inductive types*, which are recursive data types well-suited to encode elementary mathematical and computer science structures. For example, a tree could be defined as

```
type tree =
| Leaf
| Node of tree * tree
```

and an example of an object of type tree would be `Node(Node(Leaf,Leaf),Leaf)`.

Ocaml has *strong* typing. This means two things. First, every object has a type. Secondly, in a valid Ocaml program, the only way that the object `f x` is a valid object of type `B` is if the function `f` can be given the type `A -> B` and if `x` has the type `A`. Although this may seem a trivial property, many languages do not have it. For example in Python, the `math.exp` function has type `float -> float`, but `math.exp('foo')` is accepted by the Python compiler in a function definition. However, if this line of code is ever executed it will result in a failure at runtime.

Typing in Ocaml is also *static*, meaning that programs are typechecked as they are compiled and not at runtime. If the compiler accepts an Ocaml program, then it must be well typed.

Experience shows that strong and static typing, which were initially motivated by the theorem proving usecase of ML, allow for the compiler to automatically detect many of the mistakes which are easy to make in more loosely typed or non-typed programming languages.

Another distinguishing aspect of Ocaml is that it is a functional programming language: functions are first-class objects, and they can be given as input or returned as outputs of programs. On top of this, Ocaml also has more common features such as imperative primitives (`while` and `for` loops) and an object-oriented paradigm.

1.4.2 Coq

Coq is a proof assistant. This means that it provides a work environment which one can use to write mathematical statements and mathematical proofs of these statements. It can also be used to write programs, program specifications (the description of the properties a program's outputs should have) and proofs that a program satisfies a given specification.

Coq is based on a formal language called the Calculus of Inductive Constructions. This language consists of objects called *terms*, which are assigned a *type*. Types are themselves terms, and as such they themselves have a type. This is unlike Ocaml where only base objects have types. Furthermore, types are *dependent*: they are allowed to depend on a term. For example, one can build the type of matrices $m \times n$ for given m, $n \in \mathbb{N}$.

Elementary mathematical objects such as booleans, natural numbers or rationals can be represented in Coq using inductive types (similarly to Ocaml). Through the Curry-Howard isomorphism, there is a correspondence between logical statements in intuitionistic logic and types. If `t` is a type and A is the corresponding logical statement, then a term `p` of type `t` can be seen as a proof of A. Thus, in Coq, proving the statement A amounts to building a term `p`

which typechecks to `t`: in other words, proving and programming are the same activity in this context.

The main activity of the Coq software consists in type-checking, which as explained above is how proofs are checked to be correct. Crucially, one of the typing rules in particular called the *conversion* rule allows for computations to be done inside Coq's logic. The part of Coq's code which handles type-checking is very sensitive: a bug could allow one to prove false statements. For this reason, this code is maintained in a relatively small trusted *kernel* which is kept separated from the rest of the code and to which any modification is closely examined.

As seen in Figure 1.4.2, a Coq session consists mainly of two windows: a *text editor* where one writes definitions, statements and proofs and a *goal* window where the remaining goals (i.e. statements which have not yet been proved) are listed. The part of the text highlighted in blue represents the lines of code already processed by Coq, while the remaining lines have not. When doing proofs, one typically writes and executes one line at a time.

```
File Edit Options Buffers Tools Coq Proof-General Holes Help
Save Undo
Lemma foo : forall (x : nat), x = x.        1 subgoal, subgoal 1 (ID 2)

                                              ============================
                                              forall x : nat, x = x
```

Figure 1.1: A Coq session stating: $\forall x \in \mathbb{N}, x = x$.

In order to make proofs less tedious, one can write and use procedures which partially automate the task ot theorem proving. This is done in a high-level language called Ltac which resembles Ocaml. Crucially, these tactics can only participate in *building* a proof, but are completely absent from *checking* the proof: thus they are allowed to do arbitrary modifications of the current state of the proof without hindering the trust in the kernel of Coq. Several examples of tactics, including of our own, will be described in later chapters.

More contextual information about Coq and Coq libraries will appear about the encoding of mathematical structures and the handling of computations (Part II) and about Real Analysis and tactics (Part III).

Finally, it should be noted that there exist many theorem provers, that are actively developed and used. Not all of them share the same logic as Coq; however many of them share a common ancestor called LCF. The choice of a theorem prover can have a significant impact on the way proofs are developed. The Isabelle theorem prover [101] may use classical rather than intuitionistic logic. The Lean Theorem Prover [37] is very similar to Coq, but comes with fewer mathematical libraries as it is more recent.

Part I

Experiment

Chapter 2

Introduction

When one tries to gain intuition on a mathematical problem or a conjecture, one naturally turns to a mix of mental and paper computations. In many cases, using or writing computer programs can significantly extend the range of the achievable. Computers can help *visualize* data, as in the simple case of plotting a function. They can also be used to check that a conjecture isn't trivially false. Pierre de Fermat famously conjectured that all numbers of the sequence $F_n := 2^{2^n} + 1$, for $n \in \mathbb{N}$, are prime numbers [42]. He manually checked the first five instances 3, 5, 17, 257, and 65537.

Had he had access to a computer, Fermat hardly would have missed the chance to try out the very next element of the sequence, $2^{2^5} + 1 = 4294967297$. He could have written a basic primality checking function, which runs through all integers k less than the square root of its input n to see if $n \equiv 0[k]$. The Ocaml code in Listing 2.1, once compiled, runs in less than 2 milliseconds on a standard laptop:

```
let is_prime n =
  let rec aux k =
    if k*k > n then true else
      if n mod k == 0 then (Printf.printf "not prime: %d.\n" k; false) else
        aux (k+1)
  in aux 2;;

let rec pow a = function
 | 0 -> 1
 | 1 -> a
 | n ->
 let b = pow a (n / 2) in
 b * b * (if n mod 2 = 0 then 1 else a);;

let fermat n = pow 2 (pow 2 n) + 1;;

is_prime (fermat 5);;
```

Listing 2.1: A simple program which finds a factor of F_5

It returns

```
# not prime: 641 is a factor.
```

This was in fact pointed out by Leonhard Euler [42] without the help of a

computer: “I have observed after thinking about this for many days that this number can be divided by 641”.

Attempting to disprove a conjecture with a program does not always work, if only because some conjectures may actually be true, or at least counterexamples may be harder to find. Alan Turing used a computer with only $25,600$ bits of memory to check that the first $1,104$ zeros of the Riemann zeta function were consistent with the Riemann hypothesis [68]. The latest such effort of which we are aware, by Xavier Gourdon [55], computes 10^{13} zeros of the zeta function, all consistent with Riemann's conjecture.

Computers may also be used to *come up* with conjectures. One example of this is the technique of *guessing* employed to find linear recurrence relations with polynomial coefficients satisfied by a sequence [10] [112].

This small part has two chapters. First we focus on a combinatorics conjecture which states that a square of size $2^{n-2} \times 2^{n-2}$ can be filled without overlap using tiles made up of pairs of paths of size n. We illustrate how, using the right software tool, it can be simple to toy with a nontrivial conjecture and we increase the best known value up to which this conjecture is true from $n = 5$ to $n = 7$.

Then, we consider the well-known problem of the complexity of matrix multiplication: how many arithmetic operations $C(n)$ in a field $\mathbb{K}$ does it take to multiply two square matrices of size n with coefficients in $\mathbb{K}$? For a long time, it was believed that the usual naive algorithm in $\mathcal{O}(n^3)$ was optimal. In 1969, Strassen [117] first proved that this is not the case by introducing an algorithm with complexity $\mathcal{O}(n^{\log_2 7})$. He started a race to find lower bounds on the smallest constant ω such that $C(n) = \mathcal{O}(n^\omega)$. The idea of introducing a variable ε in the algorithms by Bini [17] led to the result $\omega < 2.79$. Improvements quickly became of a purely theoretical nature: Schönhage's τ-theorem [113], which allowed for the impressive improvement to $\mathcal{O}(n^{2.55})$, is typically presented more as a result on the real number ω than as a concrete algorithm (with perhaps one exception [2]).

By contrast, in this work, we are interested in the explicit description of concrete algorithms. We obtain implementations of some of these more “theoretical” algorithms and make observations, for matrices of reasonable size. The main contributions in this chapter are of two kinds.

First, we provide a constructive proof of Schönhage's τ-theorem (Section 4.3) and introduce a new way to obtain “ε-free” algorithms to compute independent parallel matrix multiplications (Section 4.3.2), and to pre-compute their complexity without directly building them.

Secondly, we present an implementation in Ocaml of a new tool for the symbolic manipulation of tensors representing matrix product algorithms (Section 4.4). This tool allows us to actually build the tensors mentioned in the previous paragraph, and even to export them to C++ or as LaTeX pseudo-code. Our hope is that this tool can in turn be used for experiments.

Chapter 3

Experimenting with a Tiling Conjecture

In this chapter, we experiment with a conjecture submitted to us by Alin Bostan, using the Combine [110] library developed in Ocaml by Rémy El Sibaïe and Jean-Christophe Filliâtre. First we state the problem (Section 3.1). Then we describe the theory underlying the Combine library's algorithm for solving tiling problems (Section 3.2). We show the results we found (Section 3.3). Finally, we conclude (Section 3.4).

3.1 Problem Description

In the following, a *step* is a vector of $\mathbb{Z}^2$. The only steps we consider are East $(1, 0)$ and North $(0, 1)$. A *path* p is a finite sequence of steps; p is said to go from $A \in \mathbb{Z}^2$ to $B \in \mathbb{Z}^2$ if $A + \sum_{s \in p} s = B$. If p_1 and p_2 are two paths which start and end at the same points A and B of $\mathbb{Z}^2$, we call *tile* the finite area enclosed by the two paths.

The problem we consider is stated by Woan, Shapiro and Rogers [129] in the following way.

Definition 1. *A path-pair of length $n \geq 2$ is a pair of paths of $\mathbb{Z}^2$ which both start at the origin $(0, 0)$, consist of n steps and meet again for the first time after exactly n steps. All steps in these paths go either East or North. We denote by $\mathcal{P}_n$ the set of such path-pairs.*

An example of such a tile is given in Figure 3.1: The paths are respectively `NNNEEEEE` and `EEENNEEN` where the letter `N` stands for North and `E` stands for East.

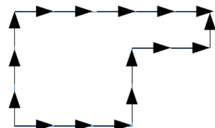
The authors go on to state and prove the following two relevant lemmas:

Lemma 1 (cardinality of $\mathcal{P}_n$). *The cardinality of $\mathcal{P}_n$ is C_{n-1}, where $(C_n)_{n \in \mathbb{N}}$ is the sequence of Catalan numbers defined by $C_n = \frac{1}{n+1}\binom{2n}{n}$.*

Lemma 2 (total area of $\mathcal{P}_n$). *The sum of the areas of the path-pairs of $\mathcal{P}_n$ is 4^{n-2}.*

The conjecture we are interested in is the following:

Figure 3.1: Example of a tile with $n = 8$



Conjecture 1 (Tiling of a square using $\mathcal{P}_n$)**.** *The figures in $\mathcal{P}_n$ can be used to tile a $2^{n-2} \times 2^{n-2}$ checkerboard, allowing rotation of the pieces by a multiple of $\frac{\pi}{2}$.*

Lemma 2 justifies that the tiling is not *completely* implausible, but aside from the fact that the areas of the tiles and the square match we are not aware of any intuition as to why Conjecture 1 may be true. However, if we draw the tiles for $n = 4$ or $n = 5$ (Figure 3.2), we see that Conjecture 1 holds for these values. The authorship of Conjecture 1 is attributed to a student called

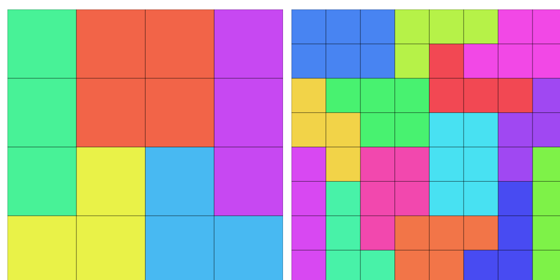


Figure 3.2: Tiling solutions for $n = 4$ and $n = 5$

Schwärzler [46]. We could not find references to solutions for $n > 5$.

3.2 Exact Matrix Cover, The Dancing Links Algorithm and Combine

In this section, we describe some of the theory underlying the Combine library and the algorithm it uses to look for tilings such as the ones displayed in Figure 3.2. First, we describe the general Exact Cover problem:

Definition 2 (Exact Cover Problem)**.** *Let X be a set and let $\mathcal{S}$ be a collection of subsets of X. An exact cover of X is a set $\mathcal{S}^* \subseteq \mathcal{S}$ such that every element of X appears in exactly one element of $\mathcal{S}^*$. In other words, $\mathcal{S}^*$ is a partition of X.*

When X is finite, finding an exact cover can be reformulated in the following way. Build a matrix M with $|X|$ columns and $|\mathcal{S}|$ rows, indexing the columns (respectively the rows) with the elements x_i of X (respectively the elements S_j of $\mathcal{S}$), so that $m_{i,j}$ is 1 if $x_i \in S_j$ and 0 otherwise. Then a set $\mathcal{S}^* \subseteq \mathcal{S}$ is a solution to the exact cover problem if and only if the sub-matrix obtained

by keeping only the rows indexed by the elements of $\mathcal{S}^*$ has exactly one 1 per column.

This reformulation is called the *exact matrix cover* (EMC) problem. It is NP-complete [48]. In 2000, Knuth published the elegant "Dancing Links" algorithm for solving this problem [79]. The algorithm's efficiency relies on the clever use of a doubly-linked-list based data-structure to represent matrices in memory.

Many well-known problems can be reformulated as instances of the EMC problem: for instance the n-queens problem (how does one place n queens on a $n \times n$ chessboard so that they don't attack one another?) or solving a Sudoku puzzle. The tiling problem of Conjecture 1 is in fact also an instance of the EMC problem.

The reformulation of the problem in Conjecture 1 as an EMC problem consists in building a matrix which describes all possible positions of the tiles on the grid. The grid G is formed of 4^{n-2} squares g. Let T be the set of tiles (in our case, $\mathcal{P}_n$) and let T' be the set of shapes which can be obtained from T: the set T' is obtained by applying all permitted rotations on T, so that each element of T' is identical to (one element of T and) at most three other elements of T', up to a rotation. The "at most" here is because duplicates due to the symmetries of a tile are removed for efficiency. To each element t' of T', we associate a set $P_{t'}$ of subsets of G. Each element of $P_{t'}$ is a legal position of t' on G.

We form a matrix M with $|G|+|T|$ columns, the 4^{n-2} first being indexed by the elements of G and the last ones by the elements of T. M has one line for each position p on G of each shape t' in T'. Hence M has $\sum_{t' \in T'} |P_{t'}|$ lines. Thus, each of the first set of columns has a 1 on every line indexed by a position which covers the corresponding square. Each of the second set of columns, indexed by a tile t, has a 1 on every line indexed by a shape obtained by a rotation of t. Each line indexed by a shape t' and a position p has a 1 on every square that is covered by putting t' at position p, and a 1 on the tile t from which it was obtained as a rotation.

Suppose we find a solution to this EMC problem. Since this solution consists in a set of lines with exactly one 1 on each column, it can be seen to be a tiling of G with the elements of T up to rotation.

The Combine library provides a powerful solver for tiling problems using the Exact Matrix Cover encoding and the Dancing Links algorithm [79].

3.3 Solutions for $n = 6$ and $n = 7$

We used Combine to confirm the conjecture for $n \leq 7$, by first generating the tiles in Ocaml for each n.

Figure 3.3 and Figure 3.4 show solutions for $n = 6$ and $n = 7$. We also show symmetric solutions found by Paul Zimmermann and his intern Romain Lemoine in Figure 3.5 and Figure 3.6. This brings a possible variation of the conjecture: for odd n, there exists a solution which is symmetric with respect to the central vertical line of the grid.

In the case of $n = 7$, the computation takes more than one hour on a standard laptop. Interestingly, when one forces the position of the two lateral bars (bottom left or bottom right) of size $n-1$, the computation only takes 20 seconds.

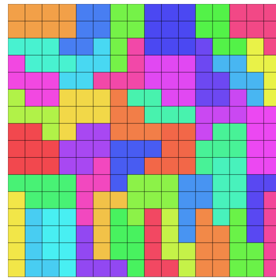


Figure 3.3: A tiling solution for $n = 6$

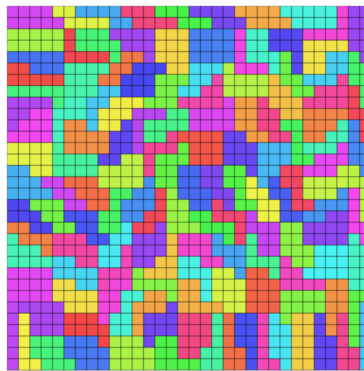


Figure 3.4: A tiling solution for $n = 7$

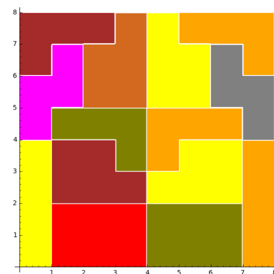


Figure 3.5: A symmetric solution for $n = 5$

In fact, there are many solutions. Although we did not count them initially, Paul Zimmermann found for example 640089600 solutions for n=5 with no constraints, 160022400 when one fixes one of the pieces which has 4 possible rotations and 2063968 if the lateral bars are fixed.

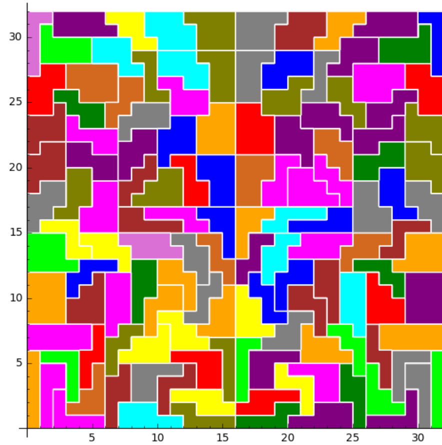


Figure 3.6: A symmetric solution for $n = 7$

3.4 Conclusion

We have improved the best known value up to which Conjecture 1 holds from $n = 5$ to $n = 7$, using the pre-existing Combine library to find tilings of a square. Unfortunately, our computer does not have enough memory for the problem to be stored in memory on our laptop for $n = 8$. A parallel approach could be justified and will be attempted later, but it would need some modifications to the Combine library.

Our point was to illustrate that often, experimenting is not only about having a computer at hand and typing a simple program such as in Listing 2.1. Having software tools tailored for specific computations can go a long way towards effectively tackling a problem.

We also tried a different (not documented here) approach for finding tilings using a linear programming encoding on a suggestion of Marc Mezzarobba, but unfortunately this turned out to yield worse times by a factor 10.

Chapter 4

Experiments with Matrix Multiplication Algorithms

It may seem like a paradox that a topic so typical of Computer Science as the complexity of multiplying matrices should serve as an illustration of how experimenting with a computer can help gain insight on results. In fact, there is a widely held belief that past some point, "fast" matrix product algorithms are a purely theoretical amusement which has no bearing on matrices of sizes typically encountered in the real world. We do not categorically dispute this notion, but the work we present here is an attempt to reconcile these theoretical results with more practical approaches to the design of algorithms.

We start by defining the problem of the complexity of Matrix Multiplication and related concepts (Section 4.1). We recall the major milestones in improving the ω constant which underlies it (Section 4.2). We present a constructive proof of Schönhage's τ-theorem; we also present a result which allows to build more practical matrix product algorithms from seemingly impractical ones (Section 4.3). This result was preceded and followed by custom software implementations for manipulating matrix product algorithms symbolically; we describe these implementations and the observations we were able to make thanks to them (Section 4.4).

4.1 Definitions

4.1.1 Matrix Multiplication

We fix a field $\mathbb{K}$. We call $\mathcal{M}_{m,n}(\mathbb{K})$ the space of matrices of size $m \times n$ over the field $\mathbb{K}$.

Let $A = (a_{i,j})_{\substack{1\leq i\leq m\\ 1\leq j\leq n}} \in \mathcal{M}_{m,n}(\mathbb{K})$ and $B = (b_{i,j})_{\substack{1\leq i\leq n\\ 1\leq j\leq p}} \in \mathcal{M}_{n,p}(\mathbb{K})$ be matrices over a field $\mathbb{K}$, of respective dimensions $m \times n$ and $n \times p$. The matrix $C = (c_{i,j})_{\substack{1\leq i\leq m\\ 1\leq j\leq p}}$ such that

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

is called the product $A \cdot B$ of A and B.

The matrix multiplication mapping

$$\begin{aligned}\langle m,n,p\rangle : \mathcal{M}_{m,n}(\mathbb{K}) \times \mathcal{M}_{n,p}(\mathbb{K}) &\longrightarrow \mathcal{M}_{m,p}(\mathbb{K}) \\ (A,B) &\mapsto A \cdot B\end{aligned}$$

is a bilinear map.

We define the complexity of a matrix multiplication algorithm $\mathcal{A}$ as the number of arithmetic operations (additions and multiplications) in $\mathbb{K}$ computed by $\mathcal{A}$. Whenever we focus on the asymptotic complexity of matrix multiplication, we will tend to only count multiplications as they asymptotically dominate additions in all the constructs we use.

It is a conjecture that the complexity of matrix multiplication is independent of $\mathbb{K}$. Until it is proved, all results are implicitly parameterized by $\mathbb{K}$. When there is an ambiguity, we will be specific about the ambient field $\mathbb{K}$.

4.1.2 Rank of a bilinear map

If U is a vector space over $\mathbb{K}$, we denote by U^* the space of linear forms over U.

Definition 3 (tensor product of two vector spaces)**.** *Let U, V be two vector spaces over $\mathbb{K}$. There exists a unique (up to isomorphism) vector space over $\mathbb{K}$, $U \otimes V$ called the tensor product of the spaces U and V, and a bilinear map*

$$\Phi : U \times V \to U \otimes V$$

such that for any vector space W over $\mathbb{K}$ and bilinear map $f : U \times V \to W$, there exists a unique linear map $f' : U \otimes V \to W$ such that $f = f' \circ \Phi$. The map Φ is denoted by $\Phi(x,y) = x \otimes y$ for $x \in U$, $y \in V$. Furthermore, $\otimes$ is associative, but not commutative.

Let U, V and W be finite dimensional vector spaces over $\mathbb{K}$ and let $f : U \times V \longrightarrow W$ be a bilinear map.

Definition 4. *The* tensor rank *of f is the smallest integer r such that there exist $x_i \in U^*, y_i \in V^*$ and $z_i \in W$ satisfying:*

$$\forall\, u \in U, v \in V,\;\; f(u,v) = \sum_{i=1}^{r} x_i(u) y_i(v) z_i. \tag{4.1}$$

We denote the tensor rank r by $R(f)$.

For any x_i, y_i and z_i satisfying Equation (4.1), the element

$$t_f = \sum_{i=1}^{r} x_i \otimes y_i \otimes z_i$$

of the space $U^* \otimes V^* \otimes W$ is called a tensorial representation of f.

Let us fix bases for U, V and W. Up to the canonical isomorphism between U and U^* (respectively V and V^*, W and W^*), t_f can be seen as an element of $U \otimes V \otimes W$, a product which gives a similar role to U, V and W. Thus, more generally, we define the tensor rank $R(t)$ of $t \in U \otimes V \otimes W$ as the minimum

r such that t can be written as a sum of r elementary tensors $x_i \otimes y_i \otimes z_i$, i.e. that t can be written

$$t = \sum_{i=1}^{r} x_i \otimes y_i \otimes z_i.$$

with $x_i \in U$, $y_i \in V$ and $z_i \in W$ for $1 \leq i \leq r$. It is easy to see that for any t_f as in Definition 4, $R(t_f) = R(f)$ by definition, so that we can now indifferently speak of the tensor rank of f or the tensor rank of t_f.

Note that the tensor rank of f or t depends on the field $\mathbb{K}$ and should be written $R_{\mathbb{K}}(f)$ or $R_{\mathbb{K}}(t)$; when there is no ambiguity we will keep the lighter notation $R(f)$ and $R(t)$.

Example: Identifying row and column matrices with $\mathbb{K}^n$, the mapping

$$\begin{aligned} \langle 1,2,1\rangle : & \mathbb{K}^2 \times \mathbb{K}^2 \to \mathbb{K} \\ & ((u,v),(x,y)) \mapsto u \cdot x + v \cdot y \end{aligned}$$

has tensor rank ≤ 2. Indeed, if we take $\pi_1 : \mathbb{K}^2 \to \mathbb{K}$ to be the first projection $(u,v) \mapsto u$ and $\pi_2 : \mathbb{K}^2 \to \mathbb{K}$ to be the second projection $(u,v) \mapsto v$, then if $A,\, B \in \mathbb{K}^2$

$$\langle 1,2,1\rangle (A,B) = \pi_1(A)\pi_1(B) \begin{pmatrix} 1 \end{pmatrix} + \pi_2(A)\pi_2(B) \begin{pmatrix} 1 \end{pmatrix}.$$

In fact, as soon as $|\mathbb{K}| \geq 2$, $R_{\mathbb{K}}(\langle 1,2,1\rangle) = 2$.

The particular case of the matrix multiplication

Computing the tensor rank of $\langle m,n,p\rangle$ gives a hint of the complexity of matrix multiplication. In fact, it can be seen that the tensor rank of $\langle n,n,n\rangle$ is less than twice the arithmetic complexity of multiplying two $n \times n$ matrices in $\mathbb{K}$. This is proved for example in a book by Burgisser *et al.* [22].

Each algorithm computing the matrix multiplication $\langle m,n,p\rangle$ that we consider corresponds to a particular way of writing $\langle m,n,p\rangle$ as a sum of elementary tensors. For instance, the naive matrix product algorithm can be written

$$\sum_{i,j,k} a_{i,k} \otimes b_{k,j} \otimes c_{i,j} = \langle m,n,p\rangle \tag{4.2}$$

so that in particular

$$R(\langle m,n,p\rangle) \leq mnp. \tag{4.3}$$

The substitution of tensor rank for arithmetic complexity leads us to define the following constant:

$$\omega := \inf\{\tau \in \mathbb{R} \,|\, R(\langle n,n,n\rangle) = \mathcal{O}(n^{\tau})\}.$$

In this part, we are thus interested in minimizing the rank $R(\langle n,n,n\rangle)$. Equation (4.3) yields a first upper bound for ω, as the sum has mnp terms, i.e n^3 if $m = n = p$:

$$\omega \leq 3. \tag{4.4}$$

Obtaining a lower bound requires introducing more technical material outside of the scope of the present text, but the reader can refer to the book [22] by Burgisser *et al.* to see how to obtain:

$$\omega \geq 2. \tag{4.5}$$

Let us consider some aspects of the tensor notation as regards matrix multiplication algorithms.

First, consider the tensor product $\otimes$ of two tensors: the object $\langle m_1, n_1, p_1\rangle \otimes \langle m_2, n_2, p_2\rangle$ can be seen as the tensor obtained when associating to each of the elementary tensors of $\langle m_1, n_1, p_1\rangle$ a copy of the tensor $\langle m_2, n_2, p_2\rangle$. This is exactly what we do when we perform a block matrix product: for a moment, we see each block as a coefficient, only to remember later that it is actually a matrix.

It should hence be no surprise that for all $m_1, n_1, p_1, m_2, n_2, p_2$ we have:

$$\langle m_1, n_1, p_1\rangle \otimes \langle m_2, n_2, p_2\rangle \cong \langle m_1\, m_2, n_1\, n_2, p_1\, p_2\rangle, \tag{4.6}$$

where the meaning of $\cong$ is the following: let $\underline{n}$ denote $\{1,\cdots,n\}$. There exists a bijective function

$$f : \big(\underline{m_1} \times \underline{n_1} \times \underline{p_1}\big) \times \big(\underline{m_2} \times \underline{n_2} \times \underline{p_2}\big) \longrightarrow \underline{m_1 m_2} \times \underline{n_1 n_2} \times \underline{p_1 p_2}$$
$$((i_1, j_1, k_1), (i_2, j_2, k_2)) \mapsto (m_2(i_1 - 1) + i_2, n_2(j_1 - 1) + j_2, p_2(k_1 - 1) + k_2)$$

such that if we replace $a_{i,k} \otimes b_{k,j} \otimes c_{i,j} \otimes a_{i_1,k_1} \otimes b_{k_1,j_1} \otimes c_{i_1,j_1}$ with $a_{i_2,k_2} \otimes b_{k_2,j_2} \otimes c_{i_2,j_2}$, where $(i_2, j_2, k_2 = f((i,j,k),(i_1,j_1,k_1))$, then the tensor product on the left becomes the tensor on the right.

Another important operation for our purposes is taking a *direct sum* of tensors. Let $t_1 \in U_1 \otimes V_1 \otimes W_1$ and $t_2 \in U_2 \otimes V_2 \otimes W_2$. Then the direct sum $t_1 \oplus t_2$, which lives in the space $(U_1 \otimes V_1 \otimes W_1) \oplus (U_2 \otimes V_2 \otimes W_2)$ can be interpreted as the independent parallel computation of t_1 and t_2. We will denote the sum $\underbrace{t \oplus t \cdots \oplus t}_{s \text{ times}}$ by $s \odot t$. In practice, as long as t_1 and t_2 do not share any variables, we will simply obtain $t_1 \oplus t_2$ as $t_1 + t_2$.

Algorithmic interpretation: In this context, any elementary tensor can be seen as a description of a program computing a matrix product, where the first two components of the tensor provide the coefficients to be multiplied, and the third one, a weighted list of the result matrix; i.e. the one where the results will eventually be stored.

Hence the elementary tensor $(a_{1,2} + a_{3,5}) \otimes b_{2,4} \otimes (c_{1,4} + 2 \cdot c_{2,4})$ reads as the algorithm

$$\begin{aligned} tmp &\leftarrow (a_{1,2} + a_{3,5}) \cdot b_{2,4} \\ c_{1,4} &\leftarrow tmp \\ c_{2,4} &\leftarrow 2 \cdot tmp \end{aligned}$$

We consider a few important properties of the tensor rank, which also justify why it is called this way.

4.1.3 Usual properties of the rank

The following properties of the rank are stated without their proofs, which can be found for example in Burgisser *et al.* [22]. We will give an intuition of what these properties mean in the context of tensors representing matrix multiplication algorithms.

Definition 5. *Let π be a permutation of $\{1,2,3\}$ and $t \in U_1 \otimes U_2 \otimes U_3$. We define $\pi(t) \in U_{\pi^{-1}(1)} \otimes U_{\pi^{-1}(2)} \otimes U_{\pi^{-1}(3)}$ to be the corresponding permutation of components on elementary tensors.*

We have $\pi(\langle m_1, m_2, m_3\rangle) = \left\langle m_{\pi^{-1}(1)}, m_{\pi^{-1}(2)}, m_{\pi^{-1}(3)}\right\rangle$.

Lemma 3. *Let $f : U \times V \longrightarrow W$ and $f' : U' \times V' \longrightarrow W'$ be bilinear maps, and $t \in U \otimes V \otimes W$, then we have:*

- $R(f \oplus f') \leq R(f) + R(f')$;
- $R(f \otimes f') \leq R(f)\ R(f')$.
- $R(\pi(t)) = R(t)$

The first item expresses that it is not more complicated to compute the sum of two independent tensors than to compute one and then the other. The second item simply states that the number of terms in the expansion of the product of two sums is the product of the number of terms in each sum. In terms of matrix multiplication, it implies that if we can compute $\langle m_1, n_1, p_1\rangle$ in r_1 multiplications and $\langle m_2, n_2, p_2\rangle$ in r_2 multiplications, then we can compute $\langle m_1m_2, n_1n_2, p_1p_2\rangle$ in r_1r_2 multiplications: divide matrices into blocks so that you can do r_1 block products, and do every product between two blocks using r_2 multiplications.

Finally, we prove the third point in the particular case of interest for us, that of matrix multiplication. Consider the trace trilinear form $(A,B,C) \mapsto Tr(ABC)$ on matrices of sizes (m,n), (n,p) and (p,m) for some positive integers m, n and p. Notice that for $1 \leq i \leq m$ and $1 \leq j \leq p$, the coefficient of $c_{j,i}$ in $Tr(ABC)$ (seen as a polynomial expression) is the matrix coefficient of index (i,j) of the matrix product AB (corresponding to $\langle m,n,p\rangle$). In the same way, the coefficient of $b_{k,j}$, $1 \leq k \leq n$, $1 \leq j \leq p$ in $Tr(CAB)$ is the matrix coefficient of index (j,k) of the product CA (corresponding to $\langle p,m,n\rangle$). A similar property holds for $Tr(BCA)$. Any permutation of $\langle m,n,p\rangle$ can be obtained, possibly by transposing A, B or C. By observing that $Tr(ABC) = Tr(CAB) = Tr(BCA)$, we can then convert any explicit tensor for $\langle m,n,p\rangle$ into a tensor for a permutation of $\langle m,n,p\rangle$ by a simple syntactic rewriting which does not change the number of terms, thus all six permutations have the same rank.

Lemma 4.

$$R(\langle n,n,n\rangle) \leq r \Rightarrow n^{\omega} \leq r \Rightarrow \omega \leq \log_n(r)$$

This a consequence of the second item of Lemma 3. From Lemmas 3 and 4 we can deduce

Lemma 5.

$$R(\langle m,n,p\rangle) \leq r \Rightarrow (mnp)^{\omega} \leq r^3.$$

Proof. Let π be the cyclic permutation mapping 1 to 2, 2 to 3, and 3 to 1.

$$\langle mnp, mnp, mnp\rangle \cong \langle m, n, p\rangle \otimes \pi\left(\langle m, n, p\rangle\right) \otimes \pi^2\left(\langle m, n, p\rangle\right)$$

so that using the second and third items of Lemma 3,

$$R(\langle mnp, mnp, mnp\rangle) \leq r^3$$

which yields the result by Lemma 4.

□

4.1.4 Degeneration and border rank

The notion of *border rank* was crucial in improving the upper bound on ω. The general idea is to relax tensors to be, not just a particular way of writing a matrix product algorithm, but some perturbation of one containing a special variable ε. Such algorithms are usually called *approximation algorithms*.

In the following, U, V, W designate vector spaces over $\mathbb{K}$.

Definition 6. *We define $U^{\mathbb{K}[\varepsilon]}$ to be the extension of scalars of U to the ring $\mathbb{K}[\varepsilon]$.*

From now on, we will use the notation T for $U \otimes V \otimes W$ (whose elements will be called plain tensors) and T_ε for $U^{\mathbb{K}[\varepsilon]} \otimes V^{\mathbb{K}[\varepsilon]} \otimes W^{\mathbb{K}[\varepsilon]}$ (whose elements will be called tensors with ε's or polynomial tensors).

The following definition may seem a bit dry, but the idea is actually simple: suppose the plain tensor we are trying to write as a minimal sum of elementary tensors is $t_1 \in T$. The previous problem we considered until now was to find a way to directly write t_1 as a sum of elementary tensors. Now, we relax this by finding some tensor $t_2 \in T_\varepsilon$ such that t_1 is the "tensor of lowest degree of t_2":

$$t_2 = \varepsilon^{q-1} t_1 + \varepsilon^q t_3$$

where $t_3 \in T_\varepsilon$.

Definition 7. *Let $t_1 \in T$ and $t_2 \in T_\varepsilon$. We say that t_2 is a degeneration of order q of t_1 (denoted $t_1 \trianglelefteq_q t_2$) if there exists $t_3 \in T_\varepsilon$ such that*

$$t_2 = \varepsilon^{q-1} t_1 + \varepsilon^q t_3.$$

The border rank $\underline{R}(t)$ *of $t \in T$ is the smallest r such that there exist $u_i \in U^{\mathbb{K}[\varepsilon]}, v_i \in V^{\mathbb{K}[\varepsilon]}, w_i \in W^{\mathbb{K}[\varepsilon]}$ and $q \geq 1$ such that:*

$$t \trianglelefteq_q \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i.$$

When there is no ambiguity, we will leave q implicit and write $t_1 \trianglelefteq t_2$.

We want to show that we can extract a tensor of low rank computing a tensor t using such a degeneration of t. First we state that the degree q of approximation does not grow too fast (linearly) when we take N-th powers:

Lemma 6. *Let $t \in T$, $t_1 \in T_\varepsilon$ and $N \in \mathbb{N}$.*

$$t \trianglelefteq_q t_1 \Rightarrow t^{\otimes N} \trianglelefteq_{N(q-1)+1} t_1^{\otimes N}.$$

This enables us to establish a result analogous to Lemma 4, but *for tensors with ε's*:

Lemma 7. *Suppose*

$$\underline{R}\left(\langle m,n,p\rangle\right) \leq r.$$

Then

$$\omega \leq \frac{3\log(r)}{\log(mnp)}.$$

This means that if we can find a degeneration of a matrix product with r (polynomial) multiplications, it yields the *same* bound on ω as if it were a plain tensor. The proof of Lemma 7 relies on the following lemma:

Lemma 8. *Let $t \in T$ and $r \in \mathbb{N}$. Then if $\underline{R}(t) \leq r$,*

$$R(t) \leq r\frac{q(q+1)}{2}$$

Proof. Suppose $\underline{R}(t) \leq r$, and write t $\trianglelefteq_q \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$ as in Definition 7 (with possibly some terms equal to zero if $\underline{R}(t) < r$). Expand each elementary polynomial tensor $u_i(\varepsilon) \otimes v_i(\varepsilon) \otimes w_i(\varepsilon)$ into a polynomial in ε with coefficients in $U \otimes V \otimes W$. Keep only the terms of degree $q-1$: there are at most as many as there are triples of nonnegative integers (i,j,k) whose sum is $q-1$ (because each monomial is built from the product of three monomials in ε), which is $\frac{q(q+1)}{2}$. The tensor obtained by removing terms of higher degree is $\varepsilon^{q-1}t$, and it can be written as the sum of at most $r\frac{q(q+1)}{2}$ elementary tensors, so that

$$R(t) \leq r\frac{q(q+1)}{2}.$$

□

We can now conclude the proof of Lemma 7: $\underline{R}\left(\langle m,n,p\rangle\right) \leq r$ so that $\underline{R}\left(\langle m,n,p\rangle^{\otimes N}\right) \leq r^N$ for $N \in \mathbb{N}$. Thanks to Lemma 6, we get

$$R\left(\langle m,n,p\rangle^{\otimes N}\right) \leq r^N \cdot P(N,q)$$

where $P(N,q)$ is a bivariate polynomial in N and q. Now we can apply Lemma 7 and let N tend to infinity and we get our result.

The intuition behind this result is that finding the coefficient of ε^{q-1} is asymptotically smaller than the overhead cost of computing with polynomials, since the degree of the approximation grows linearly when the rank grows exponentially with N. This is in contrast with the case of low values of N, for which this overhead is less justifiable. Thus, although border rank constitutes a formidable mathematical jump forward and allowed for spectacular improvements on ω, it is often mentioned as one of the reasons for matrix product algorithms to be deemed impractical, because extracting the coefficients only becomes faster for extremely large matrices.

4.2 A Quick Overview of Some Matrix Product Algorithms

This chapter describes some of the landmark results which have improved the best lower bound on ω from the naive estimate $\omega < 3$ (see Equation (4.4)) to the current record by Le Gall [81]: $\omega < 2.37287$. It is not exhaustive; for this we suggest a few references [116][22][2][127]. Dumas and Pan recently published a review [39] with an emphasis on *practical* matrix multiplication, which we can informally define as the multiplication of matrices of size $n \leq 10^6$.

4.2.1 Strassen's algorithm (1969)

Strassen's algorithm was the first to break the $\omega < 3$ barrier, much like Karatsuba was the first to break the barrier of $\mathcal{O}(n^2)$ for multiplying polynomials. Strassen [117] proves that

$$R(\langle 2,2,2\rangle) \leq 7$$

by exhibiting an algorithm to compute $\langle 2,2,2\rangle$ in 7 multiplications. From there, invoking Lemma 4, we get $\omega \leq \log_2(7) \approx 2.81$. Note that Winograd proposed an improvement on Strassen's algorithm which reduces the number of additions [128], but also proved that 7 multiplications is optimal.

4.2.2 Bini's Approximate Algorithms (1979)

The idea to use the border rank, presented in Section 4.1.4 was first introduced by Bini [17]. It was the first departure from the idea of finding "Strassen-like" exact matrix multiplication algorithms for a specific format with fewer multiplications than the naive algorithm. Bini introduced an approximative (i.e., using ε's) algorithm for the product

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & 0 \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \tag{4.7}$$

using 5 polynomial multiplications:

$$\begin{aligned} \phi(\varepsilon) :=& (a_{1,2} + \varepsilon a_{1,1}) \otimes (b_{1,2} + \varepsilon b_{2,2}) \otimes c_{2,1} + (a_{2,1} + \varepsilon a_{1,1}) \otimes b_{1,1} \otimes (c_{1,1} + \varepsilon c_{1,2}) - \\ & a_{1,2} \otimes b_{1,2} \otimes (c_{1,1} + c_{2,1} + \varepsilon c_{2,2}) - a_{2,1} \otimes (b_{1,1} + b_{1,2} + \varepsilon b_{2,1}) \otimes c_{1,1} + \\ & (a_{1,2} + a_{2,1}) \otimes (b_{1,2} + \varepsilon b_{2,1}) \otimes (c_{1,1} + \varepsilon c_{2,2}) \end{aligned}$$

From this, one can easily deduce an approximative algorithm for $\langle 3,2,2\rangle$ by gluing two copies of ϕ (and changing indices appropriately). By using permutations, one can deduce approximative algorithms for $\langle 2,3,2\rangle$ and $\langle 2,2,3\rangle$, and subsequently for $\langle 12,12,12\rangle$ in $10 \cdot 10 \cdot 10 = 1000$ multiplications.

Finally, this gives

$$\underline{R}(\langle 12^n, 12^n, 12^n\rangle) \leq 1000^n$$

so that $\omega < 2.78$ by Lemma 7.

4.2.3 Schönhage's τ-theorem (1981)

In 1981, Schönhage [113] departed even further from Strassen's algorithm. While Bini had relaxed the goal of finding matrix product algorithms to that of finding *approximate* matrix product algorithms, Schönhage relaxed it even further to *approximate, independent and parallel* matrix product algorithms. The following theorem states that if we have a tensor t which is a degeneration of the *sum* of s independent matrix products of various sizes, we can deduce from it an asymptotic upper bound for the complexity of matrix multiplication of square matrices. This complexity depends on the sizes of these matrices and on the border rank of t.

Theorem 2 (Schönhage's τ-theorem)**.** *Suppose that for $s \in \mathbb{N}^*$ and for positive integers m_i, n_i, p_i, $1 \leq i \leq s$, we have*

$$\underline{R}\left(\bigoplus_{i=1}^{s} \langle m_i, n_i, p_i \rangle\right) \leq r.$$

Then, if $\beta > 0$ is such that

$$\sum_{i=1}^{s} (m_i \, n_i \, p_i)^{\beta} = r,$$

we have

$$\omega \leq 3\,\beta.$$

An equivalent statement is that

$$\sum_{i=1}^{s} (m_i \, n_i \, p_i)^{\omega} \leq r.$$

Schönhage then presents a tensor computing $\langle 4, 1, 4 \rangle \oplus \langle 1, 9, 1 \rangle$ with border rank 17, which immediately gives the corollary that $\omega < 2.55$. This tensor is presented as an illustration of Pan tables in the next section.

Although this theorem has been a substantial tool to improve bounds on ω, the usual proof is non-constructive. By this, we mean two things. First, it relies on an implicit argument of existence of a family of algorithms whose complexity tends to the resulting value of ω rather than exhibiting one. Secondly, it does not give a way to build the algorithm *concretely*. This discourages one from building them and looking at the complexity of these algorithms, and encourages the idea that the resulting complexity is only interesting *asymptotically*.

4.2.4 Pan's Trilinear aggregation (1984)

In this section, we present an idea by Pan which consists in combining disjoint matrix product tensors to take advantage of Theorem 2. The tensors are presented as rows of three-column tables, and Pan's idea consists in adding them column by column. The tensor obtained this way then needs a *correction term* to remove terms of low degree: the hope is that the rank of this correction term is low. This idea is called *trilinear aggregation* and was extensively developed in a book [100] and is more clearly and concisely recalled in [39]. Here is a simple

example of such a table. This tensor is actually from Schönhage, but we adopt a “Pan table” view of it.

$$\begin{bmatrix} x_{i,0} & y_{0,k} & \varepsilon^2 z_{k,i} \\ \varepsilon u_{0,k,i} & \varepsilon v_{k,i,0} & w_{0,0} \end{bmatrix} \tag{4.8}$$

Each line almost represents a different (and disjoint) matrix multiplication, and each column represents a component of the tensor. The *almost* is because the variables $u_{0,k,0}$ and $v_{0,i,0}$ are adjustment variables and do not correspond to actual matrix entries.

One should specify bounds, which are here $i = 0..m-1$ and $k = 0..p-1$. Thus the tensor represented by the table in Equation (4.8) reads:

$$t_1 := \sum_{i=0}^{m-1} \sum_{k=0}^{p-1} (x_{i,0} + \varepsilon u_{0,k,i}) \otimes (y_{0,k} + \varepsilon v_{k,i,0}) \otimes (\varepsilon^2 z_{k,i} + w_{0,0}). \tag{4.9}$$

One should also specify the value of the adjustment variables, under the form of linear constraints, like for instance

$$\sum_{i=0}^{m-1} u_{0,k,i} = 0 \tag{4.10}$$

$$\sum_{k=0}^{p-1} v_{k,i,0} = 0 \tag{4.11}$$

The tensor t we are interested in obtaining is the coefficient of degree 2 in ε of the tensor in Equation (4.9). In order to have $t \trianglelefteq_2 t_1$ (see Definition 7), we need to get rid of terms of degree lower than 2. Thus we add the correction term

$$\sigma = -\left(\sum_{i=0}^{m-1} x_{i,0}\right) \otimes \left(\sum_{k=0}^{p-1} y_{0,k}\right) \otimes w_{0,0}$$

and we get a tensor

$$t_2 := t_1 + \sigma \tag{4.12}$$

with $mp + 1$ multiplications in $\mathbb{K}[\varepsilon]$.

Now, looking closely at each component of the tensor, corresponding to each row of the Table (4.8)), we can see that the two matrix products we are looking at are in fact $\langle m, 1, p\rangle$ and $\langle 1, (m-1)\,(p-1), 1\rangle$.

This would be done in a naive way in $(m \cdot 1 \cdot p) + (1 \cdot (m-1)(p-1) \cdot 1) = 2\,m\,p - (m+p-1)$ elementary tensors, but t_2 in Equation (4.12) has only $mp+1$ operations. Using Theorem 2 this gives

$$(mp)^\omega + ((m-1)(p-1))^\omega \leq mp + 1 \tag{4.13}$$

For $m = p = 4$, this gives $\omega < 2.55$. Pan also proposes a three-row table which gives $\omega < 2.522$, and then improves it with other techniques leading to $\omega < 2.5167$.

4.3 A Constructive Proof of the τ-Theorem and a New Family of Algorithms

In this section, we first present a constructive proof of the τ-theorem (Theorem 2). This proof then inspires a theorem about a new family of matrix multiplication algorithms.

4.3.1 Proof

Part of the following proof is inspired from that of [2, Chapter 7]. We assume that $\underline{R}\left(\bigoplus_{i=1}^{s}\langle m_i, n_i, p_i\rangle\right) \leq r$ and we take $\beta \in \mathbb{R}$, $\beta > 0$, such that $\sum_{i=1}^{s}(m_i\, n_i\, p_i)^{\beta} = r$. Let us show that $\omega \leq 3\,\beta$. We can assume that $r > s$, unless all products are $\langle 1, 1, 1\rangle$, which is not an interesting case: it would mean we are doing s independent scalar multiplications. We will make the assumption that we are not in that case.

Recall that if t is a tensor and $s \in \mathbb{N}$, $s \odot t$ stands for $\underbrace{t \oplus t \cdots \oplus t}_{s \text{ times}}$.

We should first remark that $\otimes$ and $\oplus$ behave just like multiplication and addition up to an isomorphism. Analogously to Theorem 1, we have for $k \in \mathbb{N}$:

$$\left(\bigoplus_{i=1}^{s}\langle m_i, n_i, p_i\rangle\right)^{\otimes k} \cong \bigoplus_{\substack{\mu \\ \mu_1+\cdots+\mu_s=k}} \binom{k}{\mu} \odot \langle \prod_{i=1}^{s} m_i^{\mu_i}, \prod_{i=1}^{s} n_i^{\mu_i}, \prod_{i=1}^{s} p_i^{\mu_i}\rangle \tag{4.14}$$

Similarly to Equation (4.6), the $\cong$ sign hides a bijective rewriting function to go from left to right.

Denote $t_0 = \bigoplus_{i=1}^{s}\langle m_i, n_i, p_i\rangle$, recall that we have by hypothesis:

$$\underline{R}(t_0) \leq r$$

so that

$$\underline{R}(t_0^{\otimes k}) \leq r^k.$$

For each μ in the direct sum on the right side of Equation (4.14), let us denote by E_μ the vector space in which each term lives. Now expand $t_0^{\otimes k}$ as in Equation (4.14) and choose $\mu \in \mathbb{N}^s$ with $\mu_1 + \cdots + \mu_s = k$. If we zero out all variables which do not belong to E_μ, we get:

$$\underline{R}\left(\binom{k}{\mu} \odot \langle \prod_{i=1}^{s} m_i^{\mu_i}, \prod_{i=1}^{s} n_i^{\mu_i}, \prod_{i=1}^{s} p_i^{\mu_i}\rangle\right) \leq r^k. \tag{4.15}$$

We rename the quantities in Equation (4.15) as :

$$\underline{R}\left(S_k \odot \langle M_k, N_k, P_k\rangle\right) \leq r^k. \tag{4.16}$$

At this point, we prove the following

Lemma 9. *If $mnp > 1$ and $s < r$,*

$$\underline{R}\left(s \odot \langle m, n, p\rangle\right) \leq r \Leftrightarrow s(mnp)^{\omega} \leq r \Rightarrow \omega \leq \frac{\log(r/s)}{\log(mnp)}.$$

In fact, Lemma 9 is an easier case of the τ-theorem when all disjoint matrix products are of the same size.

Proof. The idea is to recursively compute $\langle m^k, n^k, p^k \rangle$. To this end, we see $\langle m^k, n^k, p^k \rangle$ as $\langle m^{k-1}, n^{k-1}, p^{k-1} \rangle \otimes \langle m, n, p \rangle$.

We define the sequence $(K_k)_{k \geq 1}$ by

$$K_k = \left(\frac{r}{s}\right)^{k-1} \left(mnp + r \sum_{i=1}^{k-1} \left(\frac{s}{r}\right)^i \right).$$

Notice that if $k \geq 1$, we have

$$K_{k+1} = \frac{r}{s} K_k + r.$$

We now prove by induction on $k \geq 1$ the assertion

$$H_k := \underline{R}(\langle m^k, n^k, p^k \rangle) \leq K_k.$$

- H_1 is true, since $K_1 = mnp$.
- Suppose H_{k-1} is true for some $k \geq 2$.

  Recall that $\langle m^k, n^k, p^k \rangle \cong \langle m^{k-1}, n^{k-1}, p^{k-1} \rangle \otimes \langle m, n, p \rangle$: this consists in cutting the matrix multiplication $\langle m^k, n^k, p^k \rangle$ as a higher-level matrix multiplication of size $\langle m^{k-1}, n^{k-1}, p^{k-1} \rangle$, but where the base operation is the lower-level matrix multiplication $\langle m, n, p \rangle$. The higher level multiplication can be done by induction hypothesis in less than K_{k-1} operations, each being a $\langle m, n, p \rangle$ multiplication.

  Now, we cut this set of multiplications into $\lceil \frac{K_{k-1}}{s} \rceil$ stacks of s multiplications, padding the last stack with zeroes if necessary. We get the formula:

$$\begin{aligned} \underline{R}(\langle m^k, n^k, p^k \rangle) &\leq \left\lceil \frac{K_{k-1}}{s} \right\rceil \left(\underline{R} \left(s \odot \langle m, n, p \rangle \right) \right) \\ &\leq \left(\frac{K_{k-1}}{s} + 1 \right) r = K_k. \end{aligned}$$

Our induction is proved.

Remember that $\frac{s}{r} < 1$, so that the sum in K_k can be bounded by $\frac{1}{1-\frac{s}{r}}$. Thus

$$\underline{R}(\langle m^k, n^k, p^k \rangle) \leq \left(\frac{r}{s}\right)^{k-1} \left(mnp + \frac{r}{1 - \frac{s}{r}} \right).$$

Thus by Lemma 7,

$$\frac{k\omega}{3} \leq \frac{(k-1)\log(\frac{r}{s})}{\log(mnp)} + \frac{\log\left(mnp + \frac{r}{1-\frac{s}{r}} \right)}{\log(mnp)}.$$

Dividing on both sides by k and letting k tend to infinity, we get the expected result. □

We can now use Lemma 9 to write from (4.16):

$$\omega \leq 3 \frac{\log(\frac{r^k}{S_k})}{\log(M_k N_k P_k)}. \tag{4.17}$$

Consider the equality:

$$r^k = \left(\sum_{i=1}^{s} (m_i n_i p_i)^\beta \right)^k = \sum_{\substack{\mu \\ \mu_1 + \cdots + \mu_s = k}} \binom{k}{\mu} \left(\prod_{i=1}^{s} (m_i n_i p_i)^{\mu_i} \right)^\beta . \tag{4.18}$$

The biggest term on the right is bigger than the average over the $\binom{k+s-1}{s-1} \leq (k+1)^{s-1}$ terms of the sum, so that we may choose the term $S_k \odot \langle M_k, N_k, P_k \rangle$ in Equation (4.16) such that:

$$S_k \left(M_k N_k P_k \right)^\beta \geq \frac{r^k}{(k+1)^{s-1}}. \tag{4.19}$$

However, this is not effective if we want to actually find S_k. In practice, we may choose a central multinomial coefficient as in the following lemma.

Lemma 10 (Maximal Multinomial Coefficient)**.** *Let k, $s \in \mathbb{N}^*$. Write the euclidean division of k by s, $k = qs + r$. Then $\frac{k!}{q!^{s-r}(q+1)!^r}$ is maximal among all multinomial coefficients $\binom{k}{\nu_1, \cdots, \nu_s}$.*

Proof. First, we prove that in a maximal tuple $\nu_1, \cdots, \nu_s$, all ν_i necessarily satisfy $q \leq \nu_i \leq q + 1$. Notice first that for every element $\nu_i < q$ there exists some $\nu_j \geq q + 1$. Assume for convenience that $i = 1$ and $j = 2$:

$$\frac{k!}{\nu_1! \, \nu_2! \cdots \nu_s!} < \frac{\nu_2}{\nu_1 + 1} \frac{k!}{\nu_1! \, \nu_2! \cdots \nu_s!} = \frac{k!}{(\nu_1 + 1)! \, (\nu_2 - 1)! \cdots \nu_s!}.$$

This contradicts our maximality assumption, which proves our first point.

Assume there are t values of ν_i which are equal to q, and $s - t$ which are equal to $q + 1$. Then we have

$$k = t\, q + (s - t) \, (q + 1) = k$$

so that $t = s - r$, which proves our lemma. □

In order to compute the asymptotics, we keep to Equation (4.19).

Using Equation (4.17) and Equation (4.19) we get:

$$\begin{aligned} \omega &\leq 3 \frac{\log\left((k+1)^{s-1} (M_k N_k P_k)^\beta \right)}{\log(M_k N_k P_k)} \\ &\leq 3\beta + 3(s-1) \frac{\log(k+1)}{\log(M_k N_k P_k)}. \end{aligned}$$

Since

- $\log(M_k N_k P_k) \geq \frac{1}{\beta} \log(\frac{r^k}{S_k})$ by Equation (4.19);
- $S_k = \binom{k}{\mu_1, \cdots, \mu_s} < s^k$ (by applying Theorem 1 with $x_i = 1$ for all i);

- $s < r$;

we get

$$\omega \leq 3\beta + 3(s-1)\beta \frac{\log(k+1)}{\log(\left(\frac{s+1}{s}\right)^k)}$$

and so by letting k tend to infinity, we have the desired result.

Interpretation of this result

Using the τ-theorem, spectacular theoretical improvements can be made. Equation (4.14) is mentioned as being obvious in most papers and books on the topic, but it is not so straightforward to actually write this bijection. We will not write it down here, but we programmed it in Maple and later Ocaml. We wrote a procedure Extract(t, k, μ) which takes as input a tensor t such that $\bigoplus_{i=1}^{s} \langle m_i, n_i, p_i \rangle \trianglelefteq t$, an integer $k \in \mathbb{N}$ and a vector $\mu = (\mu_1, \cdots, \mu_s) \in \mathbb{N}^s$ with $\mu_1 + \cdots + \mu_s = k$, and returns a tensor (i.e., an algorithm) t' such that

$$\binom{k}{\mu} \odot \langle \prod_{i=1}^{s} m_i^{\mu_i}, \prod_{i=1}^{s} n_i^{\mu_i}, \prod_{i=1}^{s} p_i^{\mu_i} \rangle \trianglelefteq t'.$$

Of course, if one uses a direct sum of naive matrix multiplications, one only gets $\binom{k}{\mu}$ naive products of the type $\langle M_k, N_k, P_k \rangle$. Only tensors already constituting an improvement on the naive algorithm are susceptible to yield improvements.

4.3.2 Removing ε's and counting operations

As seen in Section 4.2.4, Pan proposes a method called trilinear aggregation to build degenerate tensors representing disjoint matrix products using as few operations as possible.

Combining this idea with the above algorithm Extract associated to Schönhage's τ-theorem proves to be a fruitful approach. The following are some new definitions and propositions.

Definition 8 (Homogeneity)**.** *Let t be a disjoint matrix product tensor of length h with the $3s$ variables*

$$(x_{1,k})_{k=1..s}, (x_{2,k})_{k=1..s}, (x_{3,k})_{k=1..s}$$

the $3s$ exponents

$$(q_{1,k})_{k=1..s}, (q_{2,k})_{k=1..s}, (q_{3,k})_{k=1..s}$$

and the $3sh$ coefficients

$$(\lambda_{i,1,k})_{i=1..h,k=1..s}, (\lambda_{i,2,k})_{i=1..h,k=1..s}, (\lambda_{i,3,k})_{i=1..h,k=1..s},$$

such that

$$t = \sum_{i=1}^{h} \left(\sum_{k=1}^{s} \lambda_{i,1,k}\, x_{1,k}\varepsilon^{q_{1,k}} \otimes \sum_{k=1}^{s} \lambda_{i,2,k}\, x_{2,k}\varepsilon^{q_{2,k}} \otimes \sum_{k=1}^{s} \lambda_{i,3,k}\, x_{3,k}\varepsilon^{q_{3,k}} \right).$$

*Then we say that t is s-*homogeneous*.*

The key point is that *each variable always appears with the same exponent.* An example of a homogeneous tensor is presented in Equation (4.9).

Theorem 3. *Let t be a s-homogeneous tensor. Let $\mu = (\mu_1, \cdots, \mu_s)$, k such that $\sum_{i=1}^{s} \mu_i = k$ and let $t' = \text{Extract}(t, k, \mu)$. The $\binom{k}{\mu}$ products $\binom{k}{\mu} \odot \langle M_k, N_k, P_k\rangle :=$ t' are of the form $\varepsilon^q\, t_1$, where $q = \sum_{u=1}^{3}\sum_{i=1}^{s} q_{u,i}\mu_i$ and t_1 is an ε-free tensor (i.e, q only depends on μ).*

Proof. Let

$$\sum_{l=1}^{s} \lambda_{i_j,1,l}\; x_{1,l}\varepsilon^{q_{1,l}} \otimes \sum_{l=1}^{s} \lambda_{i_j,2,l}\; x_{2,l}\varepsilon^{q_{2,l}} \otimes \sum_{l=1}^{s} \lambda_{i_j,3,l}\; x_{3,l}\varepsilon^{q_{3,l}}, j = 1..k$$

be k terms (possibly repeated) of t. Their product appears in the expansion of $t^{\otimes k}$:

$$\bigotimes_{j=1}^{k} \left(\sum_{l=1}^{s} \left(\lambda_{i_j,1,l}\; x_{1,l}\varepsilon^{q_{1,l}}\right) \otimes \sum_{l=1}^{s} \left(\lambda_{i_j,2,l}\; x_{2,l}\varepsilon^{q_{2,l}}\right) \otimes \sum_{l=1}^{s} \left(\lambda_{i_j,3,l}\; x_{3,l}\varepsilon^{q_{3,l}}\right) \right) \tag{4.20}$$

$$= \bigotimes_{u=1}^{3} \left(\bigotimes_{j=1}^{k} \left(\sum_{l=1}^{s} \lambda_{i_j,u,l}\; x_{u,l}\; \varepsilon^{q_{u,l}} \right)\right) \tag{4.21}$$

From the $3ks$ terms of $\bigotimes_{u=1}^{3} \left(\bigotimes_{j=1}^{k} \left(\sum_{l=1}^{s} \lambda_{i_j,u,l}\; x_{u,l}\varepsilon^{q_{u,l}} \right)\right)$, the only elementary tensors which appear in t' are those of the shape $x = x_1 \otimes x_2 \otimes x_3$ with *non-zero* components of the form

$$x_u \cong \left(\bigotimes_{l=1}^{s} \left(\prod_{j \in J_l} \lambda_{i_j,u,l}\; x_{u,l}\varepsilon^{q_{u,l}} \right)^{\otimes \mu_l} \right) \tag{4.22}$$

$$\cong \varepsilon^{q_{u,1}\mu_1 + \cdots + q_{u,s}\mu_s} \bigotimes_{l=1}^{s} \left(\prod_{j \in J_l} \lambda_{i_j,u,l}\; x_{u,l} \right)^{\otimes \mu_l} . \tag{4.23}$$

As all elementary tensors of t' are of the shape (4.23), we can set $q = \sum_{u=1}^{3}\sum_{i=1}^{s} q_{u,i}\mu_i$ and factor t' by ε^q. The remaining term of this factorization is ε-free. □

Theorem 3 means that we can benefit from the τ-theorem's improvements on Pan's degenerated tensors *without the ε's* . Indeed, in the proof of the τ-theorem, we were interested in upper bounding the *border rank* $\underline{R}\,(S_k \odot \langle M_k, N_k, P_k\rangle)$. Then, we computed an asymptotic complexity for this upper bound as we knew that the *tensor rank* $R\,(S_k \odot \langle M_k, N_k, P_k\rangle)$ would be higher because of Lemma 8. However, in this case, our tensor is of the shape $\varepsilon^q t$ with t an ε-free tensor, so that by setting ε to 1, we can see that our bound on the border rank of t' is also a bound on the tensor rank of $S_k \odot \langle M_k, N_k, P_k\rangle$.

Remark 1. *In Theorem 3, the terms of t' all had the same degree in ε. If we look closely, there might be other terms of the multinomial expansion of Theorem 2 which happen to have the same degree. An improvement would be to keep, not only $\binom{k}{\mu} \odot \langle M_k, N_k, P_k \rangle$ but all terms corresponding to a $\binom{k}{\nu}$ such that $\sum_{u=1}^{3} \sum_{i=1}^{s} q_{u,i}\nu_i = \sum_{u=1}^{3} \sum_{i=1}^{s} q_{u,i}\mu_i$. One could then iterate the "τ-theorem algorithm" on the newly obtained disjoint matrix product. This might motivate a particular choice of the powers of ε in Pan's trilinear aggregating tables.*

A natural question arises after this: how do we count the number r of elementary tensors in $\mathrm{Extract}(\mathrm{t}, \mathrm{k}, \mu)$? We will be able to compute $\binom{k}{\mu}$ products of the type $\langle \prod_{i=1}^{s} m_i^{\mu_i}, \prod_{i=1}^{s} n_i^{\mu_i}, \prod_{i=1}^{s} p_i^{\mu_i} \rangle$ (where the m_i, n_i, p_i are the dimensions of each of the s disjoint matrix products) in parallel:

$$R\left(\langle 1, \binom{k}{\mu}, 1\rangle \otimes \langle \prod_{i=1}^{s} m_i^{\mu_i}, \prod_{i=1}^{s} n_i^{\mu_i}, \prod_{i=1}^{s} p_i^{\mu_i} \rangle\right) \leq r \tag{4.24}$$

so that

$$\omega \leq \frac{\log(r)}{\log\left(\binom{k}{\mu}\left(\prod_{i=1}^{s} m_i^{\mu_i} n_i^{\mu_i} p_i^{\mu_i}\right)\right)} \tag{4.25}$$

There is a way to compute r without computing t^k.

Definition 9. *Let t be a s-homogeneous tensor and let $t_i = \bigotimes_{u=1}^{3}\left(\sum_{k=1}^{s} \lambda_{i_j,1,l}\, x_{u,k}\varepsilon^{\lambda_{u,k}}\right)$ be an elementary tensor of t. A $3s$-tuple $p(t_i) \in \{0,1\}^{3\times s}$ is called a pattern of t_i if*

$$\forall 1 \leq k \leq s, 1 \leq l \leq 3, p(t_i)_{l,k} = \begin{cases} 0 & \text{if } \lambda_{i,j,k} = 0 \\ 1 & \text{otherwise} \end{cases}$$

Definition 10. *The monomial $M(p)$ associated with a pattern $p \in \{0,1\}^{3\times s}$ is*

$$\prod_{u=1}^{3} \prod_{i=1}^{s} X_i^{p_{1,i}} Y_i^{p_{2,i}} Z_i^{p_{3,i}}$$

The polynomial $P_{t_1,\cdots,t_s} \in \mathbb{Z}[X_1,\cdots,X_s,Y_1,\cdots,Y_s,Z_1,\cdots,Z_s]$ associated with a decomposition of an s-homogeneous tensor t as a sum of elementary tensors $t = \sum_{i=1}^{s} t_i$ is defined as

$$\sum_{i=1}^{s} M(p(t_i)).$$

From now on we write P_t for $P_{t_1,\cdots,t_s}$ as we fix a decomposition of t.

Theorem 4. *Let t be a s-homogeneous tensor of length s. Let $P = P_t$ be its associated polynomial. Let $k \in \mathbb{N}$ and let μ be a s-tuple of sum k. Let Q be the*

polynomial obtained by keeping only the monomials of P^k which are multiples of $R := \left(\prod_{i=1}^{s} X_i^{\mu_i}\right)\left(\prod_{i=1}^{s} Y_i^{\mu_i}\right)\left(\prod_{i=1}^{s} Z_i^{\mu_i}\right)$ *i.e.,*

$$Q = P^k - \left(P^k \mod R\right).$$

Then the number of multiplications in the algorithm yielded by Theorem 3 for N and μ is equal to the sum of the coefficients of Q (the value of Q when all its variables are set to 1).

Proof. As there is one monomial for each term of t written as a homogeneous tensor, the products of monomials $M_1 \cdots M_k$ in the expansion of P_t^k are in one-to-one correspondence with the tensor products of the elementary tensors in $t^{\otimes k}$ (see Equation (4.21)).

The operation $P^k - \left(P^k \mod R\right)$ keeps only the monomials of P^k which are multiples of the monomial R. Thus, we need to show that these monomials are exactly the ones corresponding to the elementary tensors we kept in Equation (4.21). We define $V[1] = X$, $V[2] = Y$ and $V[3] = Z$.

We kept an elementary tensor of $t^{\otimes k}$ as in Equation (4.21) if and only if one of its sub-terms had the shape of Equation (4.22). This corresponds to keeping a monomial $M_1 \cdots M_k$ if and only if for each l, among the k monomials $M_1, \cdots, M_k$, for $u \in \{1, 2, 3\}$, there are at least μ_l which correspond to a tensor containing a *non-zero* term $x_{u,l}$. The latter is equivalent to saying that for each l, there are at least μ_l monomials among $M_1, \cdots, M_k$ which are divisible by $V[u]_l$, which is equivalent to saying that the product $M_1 \cdots M_k$ is divisible by $R = (\prod_{i=1}^{s} X_i^{\mu_i})(\prod_{i=1}^{s} Y_i^{\mu_i})(\prod_{i=1}^{s} Z_i^{\mu_i})$.

□

Theorem 4 gives us an algorithm to compute the number of multiplications of the result of $\mathrm{Extract(t, k, \mu)}$ for t a s-homogeneous tensor.

4.4 Scaling the Experiment Up: Implementations from Maple to Ocaml

4.4.1 Maple Implementation: Description, Advantages and Downsides

In Section 4.3, we considered a simple question: *what do algorithms "produced" with the τ-theorem look like in practice?*

We first decided to write an implementation of the Extract algorithm in the Maple symbolic computation software. Recall that this algorithm takes as input a tensor t such that $\bigoplus_{i=1}^{s} \langle m_i, n_i, p_i \rangle \trianglelefteq t$, a natural number $k \in \mathbb{N}$, a vector $\mu = (\mu_1, \cdots, \mu_s)$ such that $\sum_{i=1}^{s} \mu_i = k$, and returns a tensor (i.e., an algorithm) t' such that

$$\binom{k}{\mu} \odot \left\langle \bigotimes_{i=1}^{s} m_i^{\mu_i}, \bigotimes_{i=1}^{s} n_i^{\mu_i}, \bigotimes_{i=1}^{s} p_i^{\mu_i} \right\rangle \trianglelefteq t'.$$

We illustrate the symbolic operations performed by this algorithm by going informally through an example with $k = 2$.

Input tensor: Suppose we start with the direct sum of two naive matrix multiplications, $\langle 2,1,2\rangle \oplus \langle 1,3,1\rangle$:

$$t := \sum_{i=0}^{1}\sum_{j=0}^{1} a_{i,0}\otimes b_{0,j}\otimes c_{i,j} + \sum_{k=0}^{2} x_{0,k}\otimes y_{k,0}\otimes z_{0,0}$$

For this example we choose $k=2$ and $\mu=(1,1)$.

Computing $t^{\otimes k}$: First, we compute $t^{\otimes 2}$. The tensor product of variables is done in the following way. Indices are computed according to the rule already exposed just after Equation (4.6), which correspond to block matrix multiplication, and names are concatenated. For example when we take the product $\sum_{i=0}^{1} a_{i,0}\otimes\sum_{k=0}^{2} x_{0,k}$, the variable a implicitly represents a matrix of size $(2,1)$ and x represents a matrix of size $(1,3)$. Thus we create a variable ax representing a matrix of size $(2\cdot 1, 1\cdot 3)=(2,3)$ and the tensor product gives $\sum_{i=0}^{1}\sum_{k=0}^{2} ax_{i,k}$. Here is $t^{\otimes 2}$:

$$\begin{aligned}&\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{i'=0}^{1}\sum_{j'=0}^{1} aa_{(2\cdot i+i'),0}\otimes bb_{0,(2\cdot j+j')}\otimes cc_{(2\cdot i+i'),(2\cdot j+j')}+\\ &\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k'=0}^{2} ax_{i,k'}\otimes by_{k',j}\otimes cz_{i,j}+\\ &\sum_{k=0}^{2}\sum_{i'=0}^{1}\sum_{j'=0}^{1} xa_{i',k}\otimes yb_{k,j'}\otimes zc_{i',j'}+\\ &\sum_{k=0}^{2}\sum_{k'=0}^{2} xx_{0,(3\cdot k+k')}\otimes yy_{(3\cdot k+k'),0}\otimes zz_{0,0}\end{aligned}$$

By grouping sums together, we can see the terms of the expansion in Equation (4.14) which we recall is:

$$\left(\bigoplus_{i=1}^{s}\langle m_i,n_i,p_i\rangle\right)^{\otimes k} \cong \bigoplus_{\substack{\mu\\ \mu_1+\cdots+\mu_s=k}} \binom{k}{\mu}\odot\langle\prod_{i=1}^{s} m_i^{\mu_i},\prod_{i=1}^{s} n_i^{\mu_i},\prod_{i=1}^{s} p_i^{\mu_i}\rangle$$

- The variable aa (respectively bb, cc) represents the vector $\mu=(2,0)$; in other words the one in which all variables come from the *first* component of the direct sum $\langle 2,1,2\rangle\oplus\langle 1,3,1\rangle$;
- The variables ax and xa (respectively by and yb, cz and zc) represent the vector $\mu=(1,1)$; in other words the one in which there is one variable (a) from the *first* component of the direct sum and one variable (x) from the *second* component of the direct sum;
- The variable xx (respectively yy, zz) represents the vector $\mu=(0,2)$.

Extracting the term corresponding to μ and renaming variables to reflect parallel independent matrix multiplications: Suppose now that we want to extract the term corresponding to $\mu = (1,1)$, which according to our expansion above (and in Equation (4.14)), should give us $\binom{2}{1,1} = 2$ independent copies of $\langle 2\cdot 1, 1\cdot 3, 2\cdot 1\rangle = \langle 2,3,2\rangle$. We need to set to zero all variables which correspond to another term. We also rename the variables in a more convenient way, with $a0$, $b0$, $c0$ for the first copy of $\langle 2,3,2\rangle$ and $a1$, $b1$, $c1$ for the second copy:

$$\sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k0=0}^{2} a0_{i,k0}\otimes b0_{k0,j}\otimes c0_{i,j} + \\ \sum_{k=0}^{2}\sum_{i0=0}^{1}\sum_{j0=0}^{1} a1_{i0,k}\otimes b1_{k,j0}\otimes c1_{i0,j0}$$

which is the output of our algorithm.

Of course, the tensor t was rather uninteresting: it consisted of two copies of a naive matrix multiplication algorithm, and we can't expect the output to be better than several copies of a naive algorithm. The main goal of this example was to explain how variables are combined, renamed and extracted depending on μ.

Let us now consider a tensor t which is a *degeneration* of $\langle 2,1,2\rangle \oplus \langle 1,3,1\rangle$, that is a tensor with ε's with strictly less elementary tensors than the naive algorithm, that is stricly less than $2\cdot 1\cdot 2 + 1\cdot 3\cdot 1 = 7$ elementary tensors. Let us apply the exact same transformations as above to $t^{\otimes 2}$, and let us call the result t'. The tensor t' is a degeneration of $2\odot\langle 2,3,2\rangle$, i.e it computes 2 matrix products of the shape $(2,3,2)$. The hope is to be able to pick sufficiently big values for k and μ, so that t' features strictly less than $2\cdot(2\cdot 3\cdot 2) = 24$ elementary tensors, or, algorithmically speaking, strictly less than 24 polynomial multiplications. Of course, in the case of our example, sticking to $k = 2$ makes finding such good values unlikely.

Symbolic computation systems are natural candidates for the implementation of such an algorithm: manipulating variables, subscripts and polynomials is one of the core use cases for such programs. We hence made a first attempt using Maple [95]. In fact, the numerous commands available in Maple to extract information from data (normalizing polynomials to see if they are equal, looking at a specific coefficient, substituting in an algebraic way) were invaluable to make quick experiments which led to the observations of Section 4.3.2.

However, there were several reasons which ultimately led us to depart from a computer algebra system like Maple to a functional programming language like Ocaml.

The Maple implementation of the τ-theorem worked really well with *concrete* instances of tensors, i.e. when the number of elementary tensors was static. When we started to do substitutions *under* a binder "$\sum_{i=1}^{k}$" for tensors represented as sums of a variable number of elementary tensors, it became hard to tame the behavior of Maple, although this may very well be due to inexperience with the latter on our part. In particular, in Pan's tensors as presented in his book [100], there is often a clean presentation such as in Equation (4.9) which is then followed by a set of rewriting rules such as in Equation (4.10). Correctly

rewriting these under a sum can prove tricky and we found it easier to write this ourself in Ocaml.

Related to the previous part is that we realized that some of what we wanted was similar to manipulations of an abstract syntax tree as it is done for instance in compilers; a statically typed functional programming language such as Ocaml is much more suited to that purpose than Maple. The abstract data structures used in Ocaml enables us to export our tensors in a modular way to a Symbolic Computation Software such as Maple or Sage, but also to C++ (in order to get actual runnable algorithms for parallel matrix computations) and to LaTeX.

4.4.2 Ocaml Implementation

We built a software tool in Ocaml enabling us to do symbolic manipulations on tensors.

Input

The input can be given this way. It consists of three parts: a list of tables, a set of constraints, and a list of the dimensions of the matrices. Each table is itself a collection of matrix multiplications (one on each row) representing Pan tables as described in Section 4.2.4. The constraints are a collection of rewriting rules of the form $x[i_1,\cdots,i_n] := e$ where x is a variable with subscripts $i_1,\cdots,i_n$ (usually $n = 2$ in our examples, but we don't constrain it) and e is an arithmetic expression of variables. Subscripts are either static natural numbers such as 0 or 1, free variables such as k, l or they can be explicit substitution such as $k := 0$. The difference between $x[0] = e$ and $x[k := 0] = e$ is that in the first case, we will only substitute the syntactic expression $x[0]$ by e but we won't substitute under the sum in $\sum_{i=0}^{1} x[i]$; whereas in the second case, we will substitute in both cases. When a variable has no subscript, it can also be written x instead of $x[]$.

Here is Schönhage's algorithm from Section 4.2.4 for instance. The table T1 corresponds to Equation (4.8) and the table T2 corresponds to

```
Table T1 :
Line l11 :
Seq([x[i,0];y[0,k];z[k,i] * eps^2],i = 0 .. (m-1),k = 0..(p-1))
Line l12 :
Seq([u[0,k,i] * eps;v[k,i,0]*eps;w[0,0]],i = 0.. m-1,k = 0..p-1)
Table T2 :
Line l21 :
[sum(x[dd,0],dd=0..(m-1));sum(y[0,ee],ee=0..(p-1));-w[0,0]]
;;
Constraints :
u[x:=0,k:=0,i] = 0
v[k,i:=0,x:=0] = 0
u[x:=0,k,i:=0] = - sum(u[0,k,dd],dd=1..m-1)
v[k:=0,i,x:=0] = - sum(v[ee,i,0],ee=1..p-1)
x[i,0] = a1[i,0]
x[dd,0] = a1[dd,0]
y[0,k] = b1[0,k]
```

```
y[0,ee] = b1[0,ee]
z[k,i] = c1[i,k]
u[0,k,i] = a2[0,(k-1)*(m-1)+(i-1)]
v[k,i,0] = b2[(k-1)*(m-1)+(i-1),0]
w[0,0] = c2[0,0]
eps = 1
;;
Spaces :
a1 b1 c1 (m,1,p)
a2 b2 c2 (1,(m-1)*(p-1),1)
;;
```

Notice that `eps = 1` is part of the constraints, where `eps` is the variable ε. This is justified by our Theorem 3. This input format is not very restrictive, and in fact we can input more classical matrix multiplication algorithms. Here is Strassen's algorithm for instance:

```
Table T1 : Line l11 : [a[1,2] - a[2,2];b[2,1] + b[2,2]; z[1,1]]
Table T2 : Line l21 : [a[2,1] - a[1,1];b[1,2] + b[1,1]; z[2,2]]
Table T3 : Line l31 : [ a[1,1];b[1,2] - b[2,2];z[2,1] + z[2,2]]
Table T4 : Line l41 : [a[2,2];b[2,1] - b[1,1];z[1,2] + z[1,1]]
Table T5 : Line l51 : [a[2,1] + a[2,2];b[1,1];z[1,2] - z[2,2]]
Table T6 : Line l61 : [a[1,2] + a[1,1];b[2,2];z[2,1] - z[1,1]]
Table T7 : Line l71 : [a[1,1]+a[2,2];b[1,1]+b[2,2];z[1,1]+z[2,2]]
;;
Constraints:
(* no constraints *)
;;
Spaces : a b c (2,2,2)
;;
```

The first seven lines, shown as one-line Pan tables represent each of the seven elementary tensors in Strassen's algorithm:

$$
\begin{aligned}
&(a_{1,2} - a_{2,2}) \otimes (b_{2,1} + b_{2,2}) \otimes z_{1,1} + \\
&(a_{2,1} - a_{1,1}) \otimes (b_{1,2} + b_{1,1}) \otimes z_{2,2} + \\
&a_{1,1} \otimes (b_{1,2} - b_{2,2}) \otimes (z_{2,1} + z_{2,2}) + \\
&a_{2,2} \otimes (b_{2,1} - b_{1,1}) \otimes (z_{1,2} + z_{1,1}) + \\
&(a_{2,1} + a_{2,2}) \otimes b_{1,1} \otimes (z_{1,2} - z_{2,2}) + \\
&(a_{1,2} + a_{1,1}) \otimes b_{2,2} \otimes (z_{2,1} - z_{1,1}) + \\
&(a_{1,1} + a_{2,2}) \otimes (b_{1,1} + b_{2,2}) \otimes (z_{1,1} + z_{2,2})
\end{aligned}
$$

Finally, the "Spaces" line expresses that a is a matrix variable of size $2, 2$, as well as b and c.

Here are the options currently permitted by our tool, which are showed when it is called with the option `--help`:

```
Tool to parse tensors in the shape of Pan's tables
-f Name of file to parse
-pow Specifies the power at which we want to compute the tensor
```

```
-output {plain|tautheorem} Type of output to compute:
plain tensor (nothing set to zero) or tau-theorem
-style {latex|cpp|maple|sage} Style of output
-raw Set to true if your tensor is not a direct sum of matrix products
-constraints Should constraints specified in <file.in> be applied?
Defaults to true
-laser Special option for the laser method, sets all parameters right
--help  Display this list of options
```

The `-raw` option decides whether we want to interpret our tensor in terms of matrix products, which is the case for all examples presented here, or if we just want to see it as a "raw" tensor, meaning a tensor not directly encoding a matrix product. This would be the case for Strassen's initial tensor for the "laser" method [118], which does not represent a direct sum of matrix products. We chose not to detail here our support for the laser method, as it is still too rudimentary for now.

Style of Output

Maple: Any tensor can be outputted to a format which will accepted as the input of a Symbolic Computation Software format. For now, only the syntax of Maple is supported, but little effort would be needed to support others such as Sage.

C++: If a tensor represents a matrix product or a direct sum of matrix products in a way that is made explicit by the spaces section, and if it doesn't have any extra variables left (such as ε in our previous examples), it can be extracted to a `C++` program computing the corresponding matrix multiplications. Of course, the latter are just necessary conditions: the correctness of the algorithm is contingent on the correctness of the tensor.

Latex: Tensors can also be exported to Latex. This feature was used to write down the examples in Section 4.4.1.

Power

One can compute powers of tensors using the `-pow` option. Composition will be done differently depending on whether the `raw` option is set to :

- `true`: To compose two variables $x_{i_1,..,i_k}$ and $y_{j_1,..,j_l}$ we concatenate both their names and indices into $xy_{i_1,..,i_k,j_1,..j_l}$;
- `false`: we compose variables in a way that reflects block matrix product, as shown in Section 4.1.2. This assumes that each variable has exactly two indices and has an associated matrix size which has been specified to the program, in the "spaces" part of the input file.

Output

The `-output` option can be either

- `plain`: in which case the computed power of the tensor will be exported as is (with no terms set to zero, possibly with ε's remaining, and with no attempt to rename variables)

- `tautheorem`: this will only be valid if `raw` is `false`, and will apply the algorithm Extract described in Section 4.4.1 to the power of the input tensor with a μ which maximizes $\binom{k}{\mu}$.

Constraints

The `constraints` option defaults to `true`, but can be set to `false` if the user does not want to take constraints specified in the input into account.

4.4.3 Exact Complexity of Some Matrix Product Algorithms

In this section, we explore exactly how many operations (scalar multiplications and additions) it takes to compute matrix products using some of the algorithms explored.

Methodology

Contrary to the number of multiplications implied by an algorithm represented by a tensor, which is straightforward (it is exactly the number of elementary tensors, which our program outputs), the number of additions is more tricky to estimate, at least given a tensor depending on parameters. Given an elementary tensor

$$(a_{1,1} + a_{1,2}) \otimes (b_{2,1} + b_{2,2}) \otimes (c_{1,1} + c_{2,2})$$

we could say there is one addition to compute $A := a_{1,1} + a_{1,2}$, one for $B := b_{2,1} + b_{2,2}$ and then one to add AB to $c_{1,1}$ and another to add it to $c_{2,2}$. If we do this naively across all elementary tensors, we might miss common terms: in $a_{1,1} \otimes (b_{2,1} + b_{2,2}) \otimes c_{1,1} + a_{1,2} \otimes (b_{2,1} + b_{2,2}) \otimes c_{2,1}$, we don't want to count the addition $b_{2,1} + b_{2,2}$ twice as it could easily be done only once. We tried two approaches for counting additions:

- The first one was to use the `optimize` function from the `codegen` library in Maple. Given a set of algebraic expressions S to compute, it gives an "optimized computation sequence", meaning a sequence of steps which will obtain all the elements of S in a hopefully small number of steps by identifying linear common subexpressions. This works well on concrete tensors, as for example when parameters such as m and p are replaced with small numeric values in the case of Schönhage's tensor in Equation (4.9);

- The second one was to statically estimate in Ocaml the number of additions, memorizing each (symbolic) encountered expression and putting it in a hashtable so that whenever we saw it again, we didn't count it again.

The downside of the first method is that it only works on relatively small tensors, so we took the minimum of both values when we had access to both, and only the Ocaml estimate otherwise.

Remark: Both methods are actually slightly overestimating the number of additions, because they are counting each variable representing the output matrix as one addition, whereas the first value stored at the initialization of the variable should not count as an addition. For example, our method applied to the naive algorithm for $\langle m, n, p\rangle$ gives mnp additions instead of $mn(p-1)$, and it gives 22 additions for Strassen's algorithm instead of 18.

Schönhage's tensor

Recall Schönhage's tensor, already presented in Equation (4.9) as an example for Pan's tables:

$$\sum_{i=0}^{(m-1)} \sum_{k=0}^{(p-1)} (u_{0,k,i} \cdot \varepsilon + x_{i,0}) \cdot (v_{k,i,0} \cdot \varepsilon + y_{0,k}) \cdot (w_{0,0} + z_{k,i} \cdot \varepsilon^2) +$$
$$(-1) \cdot \sum_{dd=0}^{(m-1)} x_{dd,0} \cdot \sum_{ee=0}^{(p-1)} y_{0,ee} \cdot w_{0,0}$$

It can be seen to be a degeneration of $\langle m, 1, p\rangle \oplus \langle 1, (m-1)(p-1), 1\rangle$ in $mp+1$ (polynomial) multiplications.

In Table A.1, we computed the 25 best values we get for ω with concrete values for m and p between 2 and 15, and by taking the tensor to a power between 2 and 4 and applying our method. For example, the last row of this table gives a "practical ω" of 2.774065. This means if one makes multiple copies of the $6 \odot \langle 49, 1764, 64\rangle$ that we obtained and packs them into a rectangular matrix product, one can build an algorithm with the corresponding ω for that rectangular matrix product. Alternatively, one can directly compute 6 matrix products of size $\langle 49, 1764, 64\rangle$ and get a speedup of $\frac{6*49*1764*64}{10308816} \simeq 3.22$ in terms of the number of multiplications.

A tensor from Pan

We wrote an input file for a tensor presented by Pan [100], which can be seen in Appendix A.2. This tensor is a degeneration of the following direct sum of matrix products:

$$\langle m-1, 1, 2p-2\rangle \oplus \langle 2, p-1, m-1\rangle \oplus \langle p-1, 2m-2, 1\rangle \tag{4.26}$$

and it consists in $2m(p+1)$ (polynomial) multiplications.

We found a mistake in the description of this tensor and had to incorporate some correcting (scalar) factors for it to be a degeneration of the disjoint matrix multiplications described in Equation (4.26). We detected this thanks to our implementation by outputting a term to Maple and trying to match it with the claimed matrix products. Once we did the correction, on top of checking the terms in Maple again, we used a testing function in Ocaml to multiply random matrices of integers to make sure we had gotten it right. The result is the tensor presented in Listing A.1, and which also comes with our distributed code.

4.5 Conclusion

In this part, we have presented the result of our experiments with programming *theoretical* matrix multiplication algorithms in Maple and Ocaml. We wrote a constructive proof of Schönhage's τ-theorem. We discovered that starting with elements of a new class of *s-homogeneous* degenerate tensors, we could build exact (ε-free) algorithms computing several matrix products of *reasonable sizes* in parallel, with significantly less multiplications than the naive algorithm and in some cases than Strassen's algorithm. We built an Ocaml tool which allows us to manipulate matrix multiplication tensors symbolically, and output them in various formats. The data structure of Pan tables is general enough that we were able to encode every matrix multiplication tensor we could get our hands on into it.

It is our hope that this tool can be of help to discover more properties of matrix multiplication tensors in the future. We have started looking at Strassen's Laser method [118] and the Coppersmith-Winograd algorithm [33] which are both crucial for more recent improvements by Williams [127] and Le Gall [81]. We do not yet have results on the last two, beyond the fact that they can be encoded as Pan tables and so they can be made into actual, concrete algorithms. To the best of our observations so far, it seems like the number of multiplications needed by these two types of algorithms for reasonable matrix sizes is *not* competitive, consistently with what is usually claimed. Nevertheless there could be a pedagogical utility to being able to manipulate such tensors.

It turns out that the ε-free independent parallel matrix product algorithms from Theorem 3 do not perform well (about 10 times worse than the naive multiplication algorithm), at least in the way we extract them to `C++`. Although this is disappointing, it need not be the end of the story. First, the way we output code to `C++` is probably far from optimal. Secondly, we hope that our tool and findings can be used to experiment and build new ε-free matrix product algorithms, just like we used the Combine library to find new tilings in Chapter 3.

Part II

Certifying Computations

Chapter 5

Introduction

The result of a computation can provide a varying degree of confidence in a mathematical fact. On the skeptical side, the output of a calculator is not enough to be convinced of a numerical fact: Ramanujan famously conjectured that $\exp(\pi\sqrt{163}) \in \mathbb{N}$ [104]. The Maple session in Listing 5.1 shows why one might believe this: using 29 digits for intermediary computations in the Maple software, it seems as though $\exp(\pi\sqrt{163}) = 262537412640768744$:

```
> Digits := 29:
> evalf(exp(Pi * sqrt(163)));
262537412640768744.00000000000
> Digits := 40:
> evalf(exp(Pi * sqrt(163)));
262537412640768743.9999999999992500725972
```

Listing 5.1: A Maple session examining Ramanujan's "almost integer"

On the more trusting side, consider the game of Sudoku which consists in filling out a 9 by 9 grid with integers so that for each row, column, or 3 by 3 square box as delimited in bold in Figure 5.1, there be exactly one instance of each integer $i \in \{1, \cdots, 9\}$. If a piece of software fills out the empty squares in a Sudoku grid with numbers, one need only do a quick visual check in order to be convinced that they are correct.

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Figure 5.1: A Sudoku grid

When possible, one would wish to obtain as much confidence as possible in outputs of computations while still benefiting from the considerable engineering feats of scientific software. In particular, this would not require to reinvent the wheel and implement a suboptimal algorithm in order to obtain better guarantees that the result was obtained in a sound way.

Proof assistants are programs in which one can manually define mathematical objects, make mathematical statements about them and write proofs of such statements. They emphasize *soundness*: a great amount of care is taken so that only valid proofs are accepted. This often comes at the expense of efficiency: even though they usually support doing computations, the latter are typically slower than equivalent code in a powerful programming language.

This power of computer algebra software comes with different assumptions. Maple will gladly simplify $\frac{x}{x} = 1$ without knowing what type of object x is or whether it can be 0 when that makes sense, and thus it is not clear what correctness might mean in most settings. Moreover, Section 12.3 will establish that regarding numerical integral computations, neither correctness (the outputted value is within some guaranteed precision of the actual value) nor termination (the program finished computing in finite time) are guaranteed.

One way to obtain more trustworthy computations would be to write programs inside a proof assistant, prove them correct, and execute them to get an indisputable answer. However, on top of the efficiency problem already mentioned, proving modern scientific software correct is a huge endeavor. A formal proof of the correctness of GMP's square root algorithm [12] took $13,000$ lines of Coq, and this is a small routine in a huge piece of software.

Not all problems have solutions which are as trivial to check as the correctness of a Sudoku board. However, many problems lie in a larger class: those for which *checking* a solution is easier than *finding* a solution. This distinction is natural if one thinks about complexity theory, whose most famous challenge consists in comparing the class $\mathcal{P}$ of problems for which there exists a deterministic algorithm which can *find* a solution in polynomial time to the larger class $\mathcal{NP}$ of problems for which there exists a deterministic algorithm which can *check* a solution in polynomial time.

Analogously, a hybrid solution for problems which lie in the right class of easily checkable solutions is to divide the work between an *oracle* and a *verifier*. A tongue-in-cheek example of this paradigm is the "twittersort" algorithm[1] for sorting arrays. When the user inputs an array of numbers to sort, the twittersort program publishes the array on the Twitter social network, and waits for someone to respond with a sorted array. If and when someone does, it checks that the numbers in the array are sorted and correspond to the input array, and if it is correct it accepts it as the final result. The oracle produces a result and a certificate.

In this paradigm, the only part which needs to be carefully proved is the specification of the work of the verifier, i.e., the verifier accepts a result if and only if it is valid. If the verifier function is not too complex, this can be done in a proof assistant so as to obtain high confidence in our results. The oracle can be implemented however one wishes, in any programming language, without any requirement of termination, completeness or correctness. This laxness is fortunate in the case of scientific software for which none of these is guaranteed.

Harrison and Théry [65] develop the idea of a *skeptic's approach* in the particular context of Computer Algebra Software (CAS). They implement an interface between the Maple CAS and the HOL proof assistant which can call the `simplify` and `factor` Maple commands.

This separation of concerns in the paradigm of certificate-based proofs pro-

[1] https://github.com/exPHAT/twitter-sort

vides an ideal combination between the power and speed of scientific computing software and the uncompromising mathematical rigor of formal proof systems. One uses each system for what it does best and avoids it for what it does worst.

The technique of *creative telescoping* is a class of algorithms which take as input recurrence relations satisfied by a summand, and produce certificates from which one can deduce recurrence relations satisfied by a sum of this summand [25]. They output a formal identity which is often easy to check, and then getting the recurrence boils down to summing a telescoping sum, that is, a sum of the shape $\sum_k (u_{k+1} - u_k)$. The easy and mechanical aspect of checking these outputted certificates may lead one to think that they constitute *proofs* of recurrence relations. A natural step would be to make these proofs into *formal proofs*, by using a proof assistant to fully verify them. In a Maple sheet [111], Salvy uses creative telescoping algorithms and in particular their implementation in the Algolib [27] Maple library to present a creative-telescoping-based proof of the following theorem in Number Theory:

Theorem 5 (Apéry, 1978). *The constant* $\zeta(3) := \sum_{i=1}^{\infty} \frac{1}{i^3}$ *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. Although Rivoal [107] and Zudilin [134] showed that at least one of the numbers $\zeta(5)$,$\zeta(7)$,$\zeta(9)$, and $\zeta(11)$ must be irrational, $\zeta(3)$ is the only one *known* to be irrational.

In the present part, we describe a formal proof of Theorem 5 inside the Coq proof assistant, using the Mathematical Components libraries [1]. This formalization follows the structure of Apéry's original proof. However, we replace the manual verification of the recurrence relations proposed by Apéry with an automatic discovery of these equations, using the symbolic computations proposed by Salvy. For this purpose, we use Maple packages to perform calculations outside the proof assistant, and we verify a posteriori the resulting claims inside Coq. By combining these verified results with additional formal developments, we obtain a complete, constructive formal proof of Theorem 5. We report on the implementation of this cooperation between a computer algebra system and a proof assistant.

We describe in detail the formalization of an upper bound on the asymptotic behavior of $lcm(1, ..., n)$, the least common multiple of the integers from 1 to n, a part of the proof which was missing in a published paper [26].

The rest of the present part is organized as follows. We first describe the background formal theories used in our development (Section 7). We then outline the proof of Theorem 5 (Chapter 6). We summarize the algorithms used in the Maple session, the data this session produces and the way this data can be used in formal proofs (Section 8.1). We then describe the proof of the consequences of Apéry's recurrence (Section 8.2). Finally, we present an elementary proof of the bound on the asymptotic behavior of the sequence $lcm(1, ..., n)$, which is used in this irrationality proof (Section 8.3), before concluding (Chapter 9).

Chapter 6

History and outline of Apéry's theorem

Van der Poorten [123] reports that Apéry's announcement of the result of Theorem 5 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 13) satisfied by two sequences (a_n) and (b_n). It took two months of collaboration between Cohen [32], Lenstra, and Van der Poorten, with the help of Zagier, to obtain a thorough proof of Theorem 5.

There exists other proofs of Apéry's theorem, such as the one by Beukers [14]. As explained for example in Fischler' survey [44], all these proofs 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:

$$\limsup_{n\to\infty} |2\delta_n|^{\frac{1}{n}} \le (\sqrt{2}-1)^4;$$

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 [63] 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 11. *Let ℓ_n be the least common multiple of integers $1 \ldots n$, then*

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

Since we still have $(\sqrt{2}-1)^4 3^3 < 1$, this observation [62, 43] is enough to conclude. Section 8.3 discusses the formal proof of Lemma 11, an ingredient which was missing at the time of writing the previous report on this work [26].

In our formal proof, we consider the pair of sequences proposed by Apéry in his proof [5, 123]:

$$a_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2, \qquad b_n = a_n z_n + \sum_{k=1}^{n} \sum_{m=1}^{k} \frac{(-1)^{m+1} \binom{n}{k}^2 \binom{n+k}{k}^2}{2m^3 \binom{n}{m} \binom{n+m}{m}} \tag{6.1}$$

where z_n denotes $\sum_{m=1}^{n} \frac{1}{m^3}$, as already used in Proposition 1.

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

Proof. Using Lemma 14, 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 not difficult 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 by combining the definitions of the sequences a and b with the study of the mysterious recurrence relation (6.2). Indeed, he made the surprising claim that Lemma 13 holds:

Lemma 13. *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:*

$$(n+2)^3 y_{n+2} - (17n^2 + 51n + 39)(2n+3) y_{n+1} + (n+1)^3 y_n = 0. \tag{6.2}$$

Equation 6.2 is a typical example of a linear recurrence equation with polynomial coefficients and standard techniques [111, 123] 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 last two properties of our criterion, and conclude with the irrationality of $\zeta(3)$. Our formal proof relies on an elementary version of this asymptotic study, mostly based on variations on the presentation of van der Poorten [123]. We detail this part of the proof in Section 8.2.

Using only Equation 6.2, and 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 6.1.

The proof of Lemma 13 was by far the most difficult part in Apéry's original exposition. In his report [123], 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. The proof of Lemma 13 boils down 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 $k > n$):

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

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

$$U_{n,k} - U_{n,k-1} = (n+1)^3\lambda_{n+1,k} - (34n^3+51n^2+27n+5)\lambda_{n,k} + n^3\lambda_{n-1,k} \quad (6.3)$$

Summing Equation 6.3 on k shows that the sequence a satisfies the recurrence relation of Lemma 13. A similar calculation proves the analogue for b, using telescoping sums of the sequence V.

Not only is the statement of Formula 6.2 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 13 than by exhibiting this nature of certificates. Fortunately, the sequences a and b belong in fact to a class of objects well known to combinatorialists and computer-algebraists, called ∂-finite sequences. Following seminal work of Zeilberger's [131], algorithms have been designed and implemented in computer-algebra systems, which are able to obtain linear recurrences for these sequences. For instance the Maple package Mgfun (distributed as part of the Algolib [27] library) implements these algorithms, among others. Salvy's worksheet [111] is based on this implementation and 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. Our formal proof of Lemma 13 follows an approach similar to the one of Salvy. It is based on calculations performed using the Algolib [27] library, and certified a posteriori. This part of the formal proof is discussed in Section 8.1.

Chapter 7

Preliminaries

This section provides some hints about the representation of the different natures of numbers at stake in this proof in the libraries backing the formal development. It also describes a few extensions devised for these libraries and fixes some notations used throughout the present part. Most of the material presented here is related to the MATHEMATICAL COMPONENTS libraries [1] [52].

7.1 Integers

In COQ, the set $\mathbb{N}$ of natural numbers is usually represented by the type `nat`:

```
Inductive nat := O | S : nat -> nat.
```

This type is defined in a prelude[1] library, which is automatically imported by any COQ session. It models the elements of $\mathbb{N}$ using a unary representation: COQ's parser reads the number 2 as the term `S (S O)`. This inductive type comes with the usual recurrence scheme on natural numbers. This is convenient for defining elementary functions on natural numbers, like comparison or arithmetical operations, and for developing their associated theory. However, the resulting programs are usually very naive and inefficient implementations, which should only be evaluated for the purpose of small scale computations.

The set $\mathbb{Z}$ of integers can be represented by gluing together two copies of type `nat`, which provides a signed unary representation of integers:

```
Inductive int : Set := Posz of nat | Negz of nat.
```

If the term `n : nat` represents the natural number $n \in \mathbb{N}$, then the term `(Posz n) : int` represents the integer $n \in \mathbb{Z}$ and the term `(Negz n) : int` represents the integer $-(n + 1) \in \mathbb{Z}$. In particular, the constructor `Posz : nat -> int` implements the embedding of type `nat` into type `int`, which is invisible on paper because it is just the inclusion $\mathbb{N} \subset \mathbb{Z}$. In order to mimic the mathematical practice, the constant `Posz` is declared as a *coercion*, which means in particular that unless otherwise specified, this function is hidden from the terms displayed by COQ to the user (in the current goal, in answers to search queries,...) and automatically inserted.

The MATHEMATICAL COMPONENTS libraries provide formal definitions of a few concepts and results from elementary number theory, defined on the type

[1] This prelude can be seens at https://coq.inria.fr/library/Coq.Init.Prelude.html.

`nat`. For instance, they provide the theory of Euclidean division, a boolean primality test, the elementary properties of the factorial function, of binomial coefficients,... In the rest of the present part, we use the standard mathematical notations $n!$ and $\binom{n}{m}$ for the corresponding formal definition of factorial and binomial coefficients. These libraries also define the p-valuation $v_p(n)$ of a number n: if p is a prime number, $v_p(n)$ corresponds to the exponent of p in the prime decomposition of n. However, we had to extend the available formal theory with a few extra standard results like the formula giving the p-valuation of factorials:

Lemma 14 (Legendre's Formula)**.** *For any $n \in \mathbb{N}$ and for any prime number p:*

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

Incidentally, the formal version of this formula is a typical example of the slight variations one may introduce in a mathematical statement, in order to come up with a formal sentence which is not only correct and faithful to the original mathematical result, but also a tool which is easy to use in subsequent formal proofs. In the formal library, Lemma 14 is in fact stated as:

$$\text{For any prime } p \text{ and any } j, n \in \mathbb{N}, \text{ with } n < p^{j+1}, v_p(n!) = \sum_{i=1}^{j} \left\lfloor \frac{n}{p^i} \right\rfloor . \quad (7.1)$$

and the Coq counterpart is

```
Lemma fact_logp_sum_small p j n : prime p -> (n < p ^ j.+1) ->
logn p n`! =  \sum_(i < j) (n %/ p ^ i.+1).
```

Listing 7.1: The Coq statement of Equation (7.1).

Adding an extra variable to generalize the upper bound of the sum is a better option because it will ease unification when this formula is applied or used for rewriting. For example, suppose that we want to prove that $v_p(n!) = \sum_{i=1}^{\lfloor \log_p n \rfloor + 1} \left\lfloor \frac{n}{p^i} \right\rfloor$. Before applying Lemma 14, we would first need to break down the sum into two parts, and then prove that the second term $\left\lfloor \frac{n}{p^{\lfloor \log_p n \rfloor + 1}} \right\rfloor$ is zero. However, we can directly apply the lemma in Equation (7.1): Coq will unify j with $\lfloor \log_p n \rfloor + 1$ and ask us to prove $n < p^{\left(\lfloor \log_p n \rfloor + 1\right)+1}$ which is easy.

Moreover, we do not really need to introduce logarithms to state this lemma: indeed, $\lfloor \log_p n \rfloor$ is used to denote the largest power of p smaller than n. For this purpose, we could use the function `trunc_log : nat -> nat -> nat` provided by the Mathematical Components libraries, which computes the greatest exponent α such that $n^\alpha \leq m$, in other words $\lfloor \log_n m \rfloor$. Better yet, since the summand is zero when the index i exceeds this value, we can simplify the side condition on the extra variable and require only that $n < p^{j+1}$. Finally, although the fraction in the original statement of Lemma 14 may suggest that rational numbers play a role here, $\lfloor \frac{n}{m} \rfloor$ is in fact exactly the quotient of the Euclidean division of n by m. In the present part, for $n, m \in \mathbb{N}$ and m non-zero, we thus write $\lfloor \frac{n}{m} \rfloor$ for the quotient of the Euclidean division of n by m.

The basic theory of binomial coefficients present in the Mathematical Components libraries describes their role in elementary enumerative combinatorics. However, when viewing binomial coefficients as a sequence which is a certain solution of a recurrence system, it becomes natural to extend their domain of definition to integers: we thus developed a small library about these generalized binomial coefficients. We also needed to extend these libraries with properties of multinomial coefficients. We used the formula $\prod_{i=1}^{l} \binom{k_1+\cdots+k_i}{k_i}$ from Equation (1.2) in our formal definition, as it provides for free the fact that multinomial coefficients are non-negative integers, and proved the other characterizations, including the generalized Newton formula of Theorem 1 describing the expansion of $(x_1 + \cdots + x_l)^n$.

7.2 Sharing theories and notations

The Mathematical Components libraries feature a hierarchy of algebraic structures [49], which organizes a corpus of theories and notations shared by all the instances of the structures. This hierarchy implements inheritance, so that for example $\mathbb{Z}$ seen as a ring inherits from the properties of $\mathbb{Z}$ as an additive group. The hierarchy also implements sharing, so that the properties of the $+$ and $*$ operations of rings can be used in any instance of ring. Inheritance and sharing are made possible in Coq by mechanisms called coercions and canonical structures [88]. Each structure in the hierarchy is modeled by a structure called a dependent record, which packages a carrier type with some operations on this type and with requirements on these operations. For example, these structures are all *discrete*, which means that they require a boolean equality test. Part of this hierarchy deals with ordered structures [31], which means that they require an additional binary boolean predicate. This predicate has an infix notation $\leq$ and it is used to model an order relation which is (possibly partial and) compatible with the algebraic laws of the structure.

Some structures in the hierarchy feature operations that make sense only on a subset of the elements of the carrier type: for instance, by definition, only the units of a ring have an inverse. In the dependent type theory implemented by Coq, it would be possible to use a dependent pair in order to model the source type of such an inverse operation. Instead, as a rule of thumb, the signature of a given structure avoids using rich types as the source types of their operations but rather "curry" the specification. For instance, the source type of the inverse operation in the structure of ring with units is the carrier type itself, but the signature of this structure also has a boolean predicate, which selects the units in this carrier type. The inverse operation has a default behavior outside units and the equations of the theory that involve inverses are typically guarded with invertibility conditions. Hence although the expression `x^-1 * x` makes sense for any term `x` of an instance of ring with units, it can be rewritten to `1` only when `x` is known to be invertible.

In order to equip a given type with a certain structure, one should package this type with enough operations and properties, following the signature prescribed by the structure. For example, the type `int` introduced in Section 7.1 is an instance of ordered integral domain. Unlike the case of `Posz`, the canonical embedding of type `int` in any instance of ring cannot be declared as a coercion, and eluded in formal statements. Its formal definition hence comes with a

generic postfix notation `_%:~R`, used to cast an integer as an element of another ring. As for rational numbers, they are represented using a dependent pair of a pair (p, q) of integers, together with a proof that the corresponding fraction is normalized: p and q integers have to be coprime and q should be positive. By Hedberg's theorem [67], this proof can be made irrelevant by using a boolean predicate in order to express this normal form condition. Thus it is possible to compare rational numbers by comparing only the first components of the dependent pairs, and therefore to implement boolean operations of comparison. As a result, the type `rat` modeling the set $\mathbb{Q}$ of rational numbers is equipped with a structure of ordered field.

The Mathematical Components libraries also include a construction of the algebraic closure $\overline{\mathbb{Q}}$ of the field of rational numbers (more in Section 7.3), resulting in the definition of a type `algC` which is an instance of a (partially) ordered, algebraically closed field [52]. As a result, the statement `0 < x * y` makes sense in Coq whether `x` and `y` are of type `int`, `rat` or even `algC`, because all these types share the notations of the signature of ordered rings. Moreover, in all these cases, this statement can be turned into `0 < y * x` using the same lemma:

```
Lemma mulrC (R : comRingType) (r1 r2 : R) : r1 * r2 = r2 * r1.
```

because all these types are instances of the structure of commutative ring.

7.3 Algebraic numbers, real numbers

Almost all the irrational numbers involved in the proof are real algebraic numbers, and more precisely, they are of the form $r^{\frac{1}{n}}$ for some rational number r (and with $n \in \mathbb{N}$). These numbers are involved in inequalities expressing signs and estimations. It might come as a surprise that we use the type `algC` of algebraic (complex) numbers mentioned in Section 7.2 to cast these quantities, although we do not actually need complex numbers. But this choice proved convenient due to the fact that the type `algC` features both a definition of n-th roots, and a clever choice of partial order. Indeed, although $\overline{\mathbb{Q}}$ cannot be ordered as a field, it is equipped with a binary relation, denoted $\leq$, which coincides with their real order relation on $\overline{\mathbb{Q}} \cap \mathbb{R}$:

$$\forall x, y \in \overline{\mathbb{Q}},\ x \leq y \Leftrightarrow y - x \in \mathbb{R}^+.$$

In particular, for any $z \in \overline{\mathbb{Q}}$:

$$0 \leq z \Leftrightarrow z \in \mathbb{R}^+ \text{ and } z \leq 0 \Leftrightarrow z \in \mathbb{R}^-.$$

Moreover, the type `algC` comes with a function `n.-root : algC -> algC`, for any (n : nat), which computes the n-th (complex) root of its input which has a minimal non-negative argument. Crucially, when `(z : algC)` represents a non-negative real number, `(n.-root z)` coincides with the definition of the real n-th root:

```
Lemma rootC_ge0 (n : nat) (x : algC) : n > 0 -> (0 <= n.-root x) = (0 <= x).
```

The shape of Lemma `rootC_ge0` is typical of the style pervasive in the Mathematical Components libraries, where equivalences between decidable statements are stated as boolean equalities. It expresses that for an algebraic number $x \in \overline{\mathbb{Q}}$, $x^{\frac{1}{n}} \in \mathbb{R}^+$ if and only if $x \in \mathbb{R}^+$.

The one notable place at which we need to resort to a larger set of real numbers is the definition of the number $\zeta(3)$, if only because as of today, it is not even known whether $\zeta(3)$ is algebraic or transcendant. This number is defined as the limit of the sums $z_n = \sum_{m=1}^{n} \frac{1}{m^3}$, so we start our formal study by establishing the existence of this limit. In fact we formalize the part of this proof concerned with asymptotic properties using Cauchy sequences. A *Cauchy real* is a sequence $(x_n)_{n\in\mathbb{N}} \in \mathbb{Q}^{\mathbb{N}}$ together with a modulus of convergence m_x such that if $\varepsilon \in \mathbb{Q}^{+*}$, any two elements of index greater than $m_x(\varepsilon)$ are separated at most by ε.

Proposition 1. *The sequence $z_n = \sum_{m=1}^{n} \frac{1}{m^3}$ is a Cauchy real.*

```
Lemma creal_z3seq : creal_axiom z3seq.
```

Listing 7.2: Coq counterpart of Proposition 1. The predicate `creal_axiom` expresses the property of being a Cauchy real.

Two Cauchy reals x and y are equal, written $x = y$, if eventually $|x_n - y_n| < \varepsilon$, for any $\varepsilon > 0$. The Cauchy real x is smaller than y, written $x < y$, if there is an $\varepsilon > 0$ such that eventually $x_n + \varepsilon \leq y_n$. There is no effective way to compare two Cauchy reals, so these numbers cannot be equipped with the structures described in Section 7.2. We resort to the construction of Cauchy reals provided by Cohen [28] [29]. His libraries provides a type for Cauchy reals, and implements field operations for these numbers. It also provides a tactic called `bigenough`, which eases construction of the effective moduli of convergence required in the proofs that a certain property on Cauchy reals is eventually true.

7.4 Computations

In this section, we consider some aspects of computing inside Coq which are relevant to the present part, both manual and automatic.

Using the unary representation of integers described in Section 7.1, the command:

```
Compute 100*1000.
```

which asks Coq to evaluate this product, triggers a stack overflow. For the purpose of running computations inside Coq's logic, on integers of a medium size, an alternate datastructure is required, together with less naive implementations of the arithmetical operations. The present formal proof requires this nature of computations at several places, for instance in order to evaluate sequences defined by a recurrence relation at a few particular values. For these computations, we used the binary representation of integers provided by the `ZArith` library included in the standard distribution of Coq, together with the fast reduction mechanism embarked inside Coq's kernel [57].

These two ingredients are also used behind the scene by tactics implementing decision procedures. For instance, the development makes extensive use of proof commands dedicated to the normalization of algebraic expressions like the `field` and `ring` tactics [85]. The field tactic generates proof obligations describing sufficient conditions for the simplifications it made. For instance, the command `ring` might be used to automatically prove a goal of the shape `(x - 1) (x - 2) =` x^2 `- 3 x + 2`, for `x` in a ring. On the goal `(x - 1) / (x - 1) = 1`,

the command `field` would generate a side condition `x <> 1` as a new goal, and prove in the background the implication that `x <> 1 -> (x - 1) / (x - 1) = 1`.

In our case, such side conditions in turn are automatically solved using the `lia` decision procedure for linear integer arithmetic [13] so that in practice, calling `field` and then `lia` completely solves the goal if these side conditions are implied by the current proof context.

In the case of automated tactics, the conversion of the formulas in the goal into data structures suitable for larger scale computation is not visible to the user: it is performed by extra-logical code which is part of the internal implementation of these tactics. The situation is different when a computational step in a proof requires the evaluation of a formula at a given argument, and when both the formula and the argument are described using proof-oriented, inefficient representations. In that case, for instance for evaluating terms in a given sequence, the CoqEAL library [30] was used, which provides an infrastructure automating the conversion between different datastructures and algorithms used to model the same mathematical objects, like different representations of integers or different implementations of a matrix product.

Chapter 8

Formalizing Apéry's theorem in Coq

8.1 Recurrence and Creative Telescoping

Recall from Equation (6.1) that $(z_n)_{n\in\mathbb{N}}$ is the sequence $\sum_{m=1}^{n} \frac{1}{m^3}$, and that

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

Lemma 13 is the bottleneck in Apéry's proof; recall that it states that for $n \geq 0$, $(a_n)_{n\in\mathbb{N}}$ and $(b_n)_{n\in\mathbb{N}}$ both satisfy the second-order recurrence:

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

Both sums a_n and b_n are instances of *parametrised summation*: they follow the pattern $F_n = \sum_{k=\alpha(n)}^{\beta(n)} f_{n,k}$ in which the summand $f_{n,k}$, potentially the bounds, and thus the sum, depend on a parameter n. This makes it appealing to resort to the algorithmic paradigm of *creative telescoping*, which was developed for this situation in computer algebra.

To demonstrate how this works, let us look at a very simple example. Suppose we want to find a recurrence satisfied by the sequence $F_n = \sum_{k=0}^{n} \binom{n}{k}$. We start by adopting a characterization of $f_{n,k} := \binom{n}{k}$ through an implicit representation by two recurrences, which are natural if we think of the usual definition $\binom{n}{k} = \frac{n!}{k!(n-k)!}$:

$$\binom{n+1}{k} = \frac{n+1}{n+1-k} \binom{n}{k}, \qquad \binom{n}{k+1} = \frac{n-k}{k+1} \binom{n}{k}. \tag{8.1}$$

We put aside for now the question of the initial conditions which would be necessary to fully characterize them. Creative telescoping derives the following formal relation, with finite difference with respect to k on the right-hand side:

$$\binom{n+1}{k} - 2\binom{n}{k} = \left(\binom{n}{k+1} - \binom{n+1}{k+1}\right) - \left(\binom{n}{k} - \binom{n+1}{k}\right) \tag{8.2}$$

Let us check this formal identity, which is the output of an algorithm. If we expand the first part of the right-side of Equation (8.2) using the rules (8.1), we get

$$\left(\binom{n}{k+1} - \binom{n+1}{k+1}\right) - \left(\binom{n}{k} - \binom{n+1}{k}\right) = \tag{8.3}$$

$$\left(1 - \frac{n+1}{n-k}\right)\binom{n}{k+1} - \left(\binom{n}{k} - \binom{n+1}{k}\right) = \tag{8.4}$$

$$-\left(\frac{k+1}{n-k}\right)\left(\frac{n-k}{k+1}\right)\binom{n}{k} - \left(\binom{n}{k} - \binom{n+1}{k}\right) = \tag{8.5}$$

$$\binom{n+1}{k} - 2\binom{n}{k}. \tag{8.6}$$

Notice that in order to prove this, we only used the equations (8.1) so that this property is general to any bivariate sequence satisfying them.

The idea is then to sum Equation (8.2) over k from 0 to $n+1$, which gives:

$$F_{n+1} - 2F_n = \left(\binom{n}{n+2} - \binom{n+1}{n+2}\right) - \tag{8.7}$$

$$\left(\binom{n}{0} - \binom{n+1}{0}\right) \tag{8.8}$$

$$= 0 - (1-1) = 0 \tag{8.9}$$

since the right-hand side of Equation (8.2) was of the shape $v_{k+1} - v_k$. From this we get that $F_n = 2^n F_0$, so that with the right initial conditions for $f_{n,k}$, namely $f_{n,0} = 1$ and $f_{n,n} = 1$, we get the usual identity $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. This kind of proof is a typical albeit very elementary example of what the book *A=B* [102] calls *automating the discovery and proof of identities.*

Given both the magical nature of Apéry's recurrence in Lemma 13 and the painful efforts of Van der Porten, Cohen and Zagier [123] to prove it, one could try to apply creative telescoping techniques to re-discover both the recurrence and its proof *automatically*. This was done by Salvy in a Maple sheet [111] using the Algolib [27] library. From the formal proof point-of-view, the prospect of only *checking* what are essentially pre-baked proofs or rather, proof certificates, seems enticing enough.

8.1.1 Recurrences as a data structure for sequences

As already encountered in Equation (8.1), a fruitful idea from the realm of Computer Algebra is to represent sequences not explicitly, such as the univariate $(n!)_n$ or the bivariate $(\binom{n}{k})_{n,k}$, but by a system of linear recurrences with coefficients which are polynomials in the variables on which this sequence depends, accompanied with sufficient initial conditions. We borrow the following description from Chyzak [25].

A sequence $(u_{n_1,\cdots,n_r})_{n_1,\cdots,n_r \in \mathbb{N}}$ is called *hypergeometric* if for each $i \in \{1,\cdots,r\}$, there exists a rational function $R_i(n_1,\cdots,n_r)$ such that

$$u_{n_1,\cdots,n_i+1,\cdots,n_r} = R_i(n_1,\cdots,n_r) \cdot u_{n_1,\cdots,n_r},$$

in other words, if u is cancelled by the operators $R_i - S_{n_i}$ where S_{n_i} is the shift operator which maps $u_{n_1,\cdots,n_r}$ to $u_{n_1,\cdots,n_i+1,\cdots,n_r}$. For example, $\binom{n}{k}$ is hypergeometric: the recurrences (8.1) once rewritten as equalities to zero can be represented as $P \cdot f = 0$ for $P = S_n - \frac{n+1}{n+1-k}$ and $P = S_k - \frac{n-k}{k+1}$, respectively.

This notion can be generalized. First, consider a *skew* polynomial algebra $\mathcal{A} = \mathbb{Q}(n_1, \cdots, n_r) < S_{n_1}, \cdots, S_{n_r} >$ with the commutation rules

$$S_{n_1}^{i_1} \cdots S_{n_r}^{i_r} c(n_1, \cdots, n_r) = c(n_1 + i_1, \cdots, n_r + i_r) S_{n_1}^{i_1} \cdots S_{n_r}^{i_r}$$

for any rational fraction $c(n_1, \cdots, n_r)$. A polynomial

$$P = \sum_{(i_1,\cdots,i_r)} p(n_1, \cdots, n_r) S_{n_1}^{i_1} \cdots S_{n_r}^{i_r} \in \mathcal{A}$$

acts on a r-variable sequence $(u_{n_1,\cdots,n_r})$ by

$$(P \cdot u)_{n_1,\cdots,n_r} = \sum_{(i_1,\cdots,i_r)} p(n_1, \cdots, n_r) u_{n_1+i_1,\cdots,n_r+i_r},$$

where subscripts denote evaluation.

To any sequence f, one associates the set of skew polynomials that annihilate it. This set, $\{P \in \mathcal{A} : P \cdot f = 0\}$ is a left ideal of $\mathcal{A}$, named the *annihilating ideal* of f, and denoted $\operatorname{ann} f$. A hypergeometric sequence can be redefined as a function f whose quotient module $\mathcal{A}/\operatorname{ann} f$ is a vector space of dimension 1 over the rational function field $\mathbb{Q}(n_1, \cdots, n_r)$. A ∂-finite sequence is a function whose quotient module $\mathcal{A}/\operatorname{ann} f$ has finite dimension ≥ 1.

A non-commutative extension of the usual Gröbner-basis theory is available, together with algorithmic analogues. In this setting, a good representation of a ∂-finite sequence is obtained as a Gröbner basis of $\operatorname{ann} f$ for a suitable ordering on the monomials in $S_{n_1}, \cdots, S_{n_r}$. For the example of $f_{n,k} = \binom{n}{k}$, a Gröbner basis consists of both already-mentioned skew polynomials encoding (8.1). In general, a Gröbner basis provides us with a (vectorial) basis of the quotient module $\mathcal{A}/\operatorname{ann} f$, which is isomorphic to $\mathcal{A}f$. This basis can be explicitly written in the form $B = \{f_{n_1+i_1,\cdots,n_r+i_r}\}_{(i_1,\cdots,i_r)\in\mathcal{U}}$, where the finite set $\mathcal{U}$ of indices is given as the part under the classical stair shape of the Gröbner-basis theory. Given a Gröbner basis GB for $\operatorname{ann} f$, the normal form $\operatorname{NF}(p, \mathrm{GB})$ is unique for any $p \in \mathcal{A}$. Again in the binomial example, the finite set is $\mathcal{U} = \{(0, 0)\}$, and normal forms are rational functions.

This is the basis of algorithms for a number of operations under which the ∂-finite class is stable, which all operate by looking for enough dependencies between normal forms: application of an operator, addition, product. Although the following applies in the general case, we now restrict ourselves to the bivariate case which interests us more specifically from now on.

The case of summing a sequence $(f_{n,k})$ into a parametrized sum $F_n = \sum_{k=0}^{n} f_{n,k}$ is more involved: it follows the method of creative telescoping [132], in two stages. First, an *algorithmic* step determines pairs (P, Q) satisfying

$$P \cdot f = (S_k - 1) Q \cdot f \tag{8.10}$$

with $P \in \mathbb{Q}(n)[S_n]$ and $Q \in \mathcal{A}$. To continue with our example $f_{n,k} = \binom{n}{k}$, this corresponds to Equation 8.2 with $P = S_n - 2$ and $Q = 1 - S_n$. Second, a *systematic* step follows: summing (8.10) for k between 0 and $n + \deg P$ yields

$$(P \cdot F)_n = (Q \cdot f)_{k=n+\deg P+1} - (Q \cdot f)_{k=0}. \tag{8.11}$$

Continuing with our binomial example, this corresponds to the summation leading to Equation (8.7).

8.1.2 Apéry's sequences are ∂-finite constructions

The sequences a and b in (6.1) are ∂-finite: they have been announced to be solutions of (6.2). But more precisely, they can be viewed as constructed from "atomic" sequences by operations under which the class of ∂-finite sequences is stable. This is summarized in Table 8.1.

step	explicit form	GB	operation	input(s)
1	$c_{n,k} = \binom{n}{k}^2 \binom{n+k}{k}^2$	C	direct	
2	$a_n = \sum_{k=1}^{n} c_{n,k}$	A	creative telescoping	C
3	$d_{n,m} = \frac{(-1)^{m+1}}{2m^3\binom{n}{m}\binom{n+m}{m}}$	D	direct	
4	$s_{n,k} = \sum_{m=1}^{k} d_{n,m}$	S	creative telescoping	D
5	$z_n = \sum_{m=1}^{n} \frac{1}{m^3}$	Z	direct	
6	$u_{n,k} = z_n + s_{n,k}$	U	addition	Z and S
7	$v_{n,k} = c_{n,k} u_{n,k}$	V	product	C and U
8	$b_n = \sum_{k=1}^{n} v_{n,k}$	B	creative telescoping	V

Table 8.1: Construction of a_n and b_n: At each step, the Gröbner basis named in column GB, which annihilates the sequence given in explicit form, is obtained by the corresponding on the inputs given in the last column.

In this table, Gröbner bases are systems of recurrence operators: at each line in the table, the sequence given in explicit form is a solution of the system of recurrences described by the operators in the Gröbner basis column. Note that in fact none of these results rely on the *specific* sequences in the explicit form: at each step, a new Gröbner basis is obtained from known ones, the ones that are cited in the input column, just like we obtained the operator $S_{n+1} - 2S_n$ in Equation (8.7) from $S_n - \frac{n+1}{n+1-k}$ and $S_k - \frac{n-k}{k+1}$. The table can also be read bottom-up for the purpose of verification: the Gröbner basis obtained at a given step can be verified using *only* the Gröbner bases obtained at some previous steps, all the way down to C and D. These operators describe a more general class of sequences than just the explicit sequences used in this table, thus initial conditions are needed to describe a precise sequence.

8.1.3 Provisos and sound creative telescoping

Let us go back to our proof of the binomial identity in Equations (8.3) to (8.6). This proof in fact only holds if $n \neq k$ which makes taking the sum for k from 0 to $n+1$ an illegitimate operation. In another example, one can "almost prove" Pascal's triangle rule using only the recurrences (8.1):

$$\binom{n+1}{k+1} - \binom{n}{k+1} - \binom{n}{k} = \left(\frac{n+1}{n-k}\frac{n-k}{k+1} - \frac{n-k}{k+1} - 1\right)\binom{n}{k} = 0 \times \binom{n}{k} = 0,$$

but this requires $k \neq -1$ *and* $k \neq n$. Therefore, this does not prove Pascal's rule for all n and k. This incomplete modelling of sequences by algebraic objects may cast doubt on the output of creative-telescoping algorithms.

By contrast, in our formal proofs, we augmented the recurrences with provisos that restrict their applicability. In this setting, we validate a candidate identity like the Pascal triangle rule by a normalization modulo the elements of a Gröbner basis plus a verification that this normalization only involves legal instances of the recurrences. In the case of creative telescoping, Equation (8.10) takes the form:

$$(n,k) \notin \Delta \Rightarrow (P \cdot f_{_,k})_n = (Q \cdot f)_{n,k+1} - (Q \cdot f)_{n,k}, \tag{8.12}$$

where $\Delta \subset \mathbb{Z}^2$ guards the relation and where $f_{_,j}$ denotes the univariate sequence obtained by specializing the second argument of f to j. Thus our formal analogue of Eq. (8.11) takes this restriction into account and has the shape

$$\begin{aligned}(P \cdot F)_n = \Big((Q \cdot f)_{n,n+\beta+1} - (Q \cdot f)_{n,\alpha}\Big) + \sum_{i=1}^{r}\sum_{j=1}^{i} p_i(n)\, f_{n+i,n+\beta+j} \\ + \sum_{\alpha \leq k \leq n+\beta \ \wedge\ (n,k) \in \Delta} (P \cdot f_{_,k})_n - (Q \cdot f)_{n,k+1} + (Q \cdot f)_{n,k},\end{aligned} \tag{8.13}$$

for F the sequence with general term $F_n = \sum_{k=\alpha}^{n+\beta} f_{n,k}$ and where $P = \sum_{i=0}^{r} p_i(n) S_n^i$, with the p_i polynomials with rational coefficients.

The proof of identity (8.13) is a straightforward reordering of the terms of the left-hand side, $(P \cdot F)_n = \sum_{i=0}^{r} p(n)\, F_{n+i}$, after unfolding the definition of F and applying relation (8.12) everywhere allowed in the interval $\alpha \leq k \leq n + \beta$. The first part of the right-hand side is the usual difference of border terms, already present in Equation (8.11). The middle part is a collection of terms that arise from the fact that the upper bound of the sum defining F_n depends linearly on n and that we do not assume any nullity of the summand outside the summation domain. The last part, which we will call the singular part, witnesses the possible partial domain of validity of relation (8.12). The operator P is a valid recurrence for the sequence F if the right-hand side of Equation (8.13) normalizes to zero, at least outside of an algebraic locus Δ that will guard the recurrence.

The Coq counterpart to Equation (8.13) is the following

```
Lemma sound_telescoping
(cf0 : int -> R := fun _ => 0)
(r : nat := size Pseq)
(P : (int -> R) -> int -> R := horner_seqop Pseq)
(U : int -> R := fun n => \sum_(a <= k < n + b :> int) u n k)
(n : int)
(range_correct : a <= n + b)
(PeqDQ :
∀ (n k : int), not_D n k ->
P (u ^~ k) n = Q u n (k + 1) - Q u n k) :
P U n =
Q u n (n + b) - Q u n a +
\sum_(a <= k < n + b :> int | ~~ not_D n k)
(P (u ^~ k) n - (Q u n (k + 1) - Q u n k)) +
\sum_(0 <= i < degree P)
```

```
\sum_(0 <= j < i)
P_i * u (n + i) (n + j + b).
```

Listing 8.1: Our sound telescoping lemma in Coq

The notation `u ^~ k` stands for `fun n => u n k`, `P_i` denotes the i-th coefficient of the polynomial `P`. The outside of the locus Δ is represented by the predicate `not_D`.

Let us now apply the theorem of Equation (8.13) to our example of the binomial identity. The forbidden domain Δ is the singleton $\{(n, n)\}$, because that is the only point at which Equation (8.2) is not valid. We get

$$\begin{aligned}(S_n - 2)F_n = F_{n+1} - 2F_n =& ((f_{n,n+1} - f_{n+1,n+1}) + (f_{n,0} - f_{n+1,0})) + f_{n+1,n+1} + \\ & (((S_n - 2)f_{_,n})_n - (f_{n,n+1} - f_{n+1,n+1}) + (f_{n,n} - f_{n+1,n})) \\ =&(-1) + 1 + \\ & (n + 1 - 2) - (0 - 1) + (1 - (n + 1)) \\ =&0.\end{aligned}$$

The automatic nature of the process of creative telescoping takes a hit here, both because Δ has to be built manually and because it may depend on the order in which recurrences are rewritten in the proof of identity (8.12). Furthermore, we are aware of no guarantees that given a rewriting order, the various terms of Formula (8.13) will cancel as nicely as in the case of the binomial identity above.

8.1.4 Generated operators, hand-written provisos, and formal proofs

For each step in Table 8.1, we make use of the data computed by the Maple session in a systematic way. Figure 8.1 illustrates this pattern on the example of step 7. As mentioned in Section 8.1.3, we annotate each operator produced by the computer-algebra program with provisos (see below) and turn it this way into a conditional recurrence predicate on sequences. To each sequence in the program corresponds a file defining the corresponding conditional recurrences, for instance `annotated_recs_c`, `annotated_recs_u`, and `annotated_recs_v` for c, u, and v, respectively. More precisely these files contain *all* the operators obtained by the Maple script for a given sequence, not only the Gröbner basis. We use rounded boxes to depict the files that store the *definitions* of these predicates. These are generated by the Maple script which pretty-prints its output in Coq syntax, with the exception of the definition of provisos. Throughout this section, a maple leaf tags the files that are generated by our Maple script. Yet automating these annotations is currently out of reach.

In our formal proof, each step in Table 8.1 consists in proving that some conditional recurrences on a composed sequence can be proved from some conditional recurrences known for the arguments of the operation. We use square boxes to depict the files that store these *formal proofs*. The statement of the theorems proved in these files are composed from the predicates defined in the round boxes: a dashed line points to (predicates used to state) conclusions and a labelled solid line points to (predicates used to state) hypotheses.

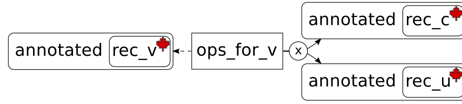


Figure 8.1: Proving that V is $C \times U$

8.1.5 Definitions of conditional recurrence predicates

All files defining the conditional recurrence predicates obtained from the operators annihilating sequences of the program share the same structure. An excerpt of the generated part of the file `annotated_recs_c` is displayed on Listing 8.2. The constants `Sn`, `Sk`, and `CT_premise` are recurrence predicates, defined in terms of a bound variable `c`. Constants `Sn` and `Sk` are elements of the Gröbner basis. The definition of these recurrences is named to reflect the term it rewrites, e.g., the left-hand sides in (8.1): these names are the result of pretty-printing the (skew) monomial that encodes these left-hand sides, the prefix `S` standing for "shift". For example `Sn` is the name of a recurrence defining $c_{n+1,k}$, while `SnSk` would be for $c_{n+1,k+1}$. Rewriting a given term with such an equation makes the term decrease for the order associated with the Gröbner basis. Another part of the file defines the recurrences obtained from a creative-telescoping pair (P, Q) generated for the purpose of the summation defining the sequence a.

```
(* Coefficients of every recurrence, P, and Q. *)
Definition Sn_cf0_0 n k := (n + 1 + k)² / (-n - 1 + k)².
Definition Sk_cf0_0 n k := (-n + k)² * (n + 1 + k)² / (k + 1)⁴.
Definition P_cf0 n := (n + 1)³.
...
(* Conditional recurrences. *)
Definition Sn c := ∀ n k, precond.Sn n k -> c (n + 1) k = Sn_cf0_0 n k * c n k
Definition Sk c := ∀ n k, precond.Sk n k -> c n (k + 1) = Sk_cf0_0 n k * c n k

(* Operators P and Q. *)
Definition P c n := P_cf0 n * c n + P_cf1 n * c (n + 1) + P_cf2 n * c (n + 2).
...
(* Statement P = Δ_k Q. *)
Definition CT_premise c := ∀ n k, precond.CT_premise n k ->
P (c ^~ k) n = Q c n (k + 1) - Q c n k.
```

Listing 8.2: Maple-generated part of annotated_rec_c

Observe that these generated definitions feature named provisos that are in fact placeholders. In the preamble of the file, displayed on Listing 8.3, we provide by a manual annotation a concrete definition for the proviso of each recurrence defined in the generated part. Observe however that part of these definitions can be inferred from the coefficients of the recurrences. For example the $k \neq n + 1$ condition in `precond.Sn`, the proviso of recurrence `Sn`, is due to the denominator $(-n - 1 + k)^2$ of the coefficient `(Sn_cf0_0 n k)`.

```
Module precond.
Definition Sn n k := (k != n + 1) ∧ (n != -1).
Definition Sk n k := (k + 1 != 0) ∧ (n != 0).
Definition CT_premise n k := (n >= 0) ∧ (k >= 0) ∧ (k < n).
End precond.
```

Listing 8.3: Hand-written provisos in annotated_rec_c

In the last part of the file, see Listing 8.4, a record collects the elements of the Gröbner basis C. Maple indeed often produces a larger set of annihilators for a given sequence, for instance `CT_premise` in Listing 8.2 is related to a creative telescoping pair but not to the Gröbner basis. Also, the Gröbner basis can be obtained by refining a first set of annihilators, which happens at step 4 of Table 8.1.

```
(* Choice of recurrences forming a Groebner basis. *)
Record Annihilators c := { Sn : Sn c; Sk : Sk c }.
```

Listing 8.4: Selection of a Gröbner basis

8.1.6 Formal proofs of a conditional recurrence

We take as a running example the file `ops_for_a`, which models step 2 in Table 8.1. This file proves theorems about an arbitrary sequence c satisfying the recurrences in the Gröbner basis displayed on Listing 8.4, and about the sequence a by definite summation over c.

```
Require Import annotated_recs_c.
Variables (c : int -> int -> rat) (ann_c : Annihilators c).

Theorem P_eq_Delta_k_Q : CT_premise c.  Proof. ... Qed.

Let a n := \sum_(0 <= k < n + 1) c n k.
```

The formal proof of lemma `P_eq_Delta_k_Q` is an instance of Equation (8.12). Using this property, we prove that the sequence a verifies a conditional recurrence associated to the operator P. As suggested in Section 8.1.3, this proof consists in applying the lemma `sound_telescoping` (see Equation (8.13) and Listing 8.1), which formalizes a sound creative telescoping, and in normalizing to zero the resulting right-hand side of Equation (8.13). Listing 8.5 displays the first lines of the corresponding proof script, which select and name the three components of the right-hand side of Equation (8.13), with self-explanatory names. The resulting proof context is displayed on Listing 8.6.

```
Theorem recApery_a n (nge2 : n >= 2) : P a n = 0.
Proof.
rewrite (punk.sound_telescoping P_eq_Delta_k_Q).
set boundary_part := (X in X + _ + _).
set singular_part  := (X in _ + X + _).
set overhead_part := (X in _ + _ + X).
```

Listing 8.5: Begining of a proof of sound creative telescoping

```
boundary_part := Q c n (n + 1) - Q c n 0
singular_part := \sum_(0 <= i < n + 1 | precond.CT_premise n i)
P (c ^~ i) n - (Q c n (i + 1) - Q c n i)
overhead_part := \sum_(0 <= i < degree P)
\sum_(0 <= j < i) P_i n * c (n + i) (n + j + 1)
============================
boundary_part + singular_part + overhead_part = 0
```

Listing 8.6: Corresponding goal

In Listing 8.6, `degree P` is the degree of `P`, two in this specific case. Note that we have access to `degree P` because in addition to the definition displayed on

Listing 8.2, we also have at our disposal a list representation `[:: P_cf0; P_cf1]` of the same operator.

The proof of the goal of Listing 8.6, proceeds in three steps. The first step is to inspect the terms in `singular_part` and to chase ill-formed denominators, like $n - n$. These can arise from the specialisations, like $k = n$, induced when unrolling the definition of (the negation of) `precond.CT_premise`. In our formalization, a division by zero is represented by a conventional value: we check that these terms vanish by natural compensations, independently of the convention, and we keep only the terms in `singular_part` that represent genuine rational numbers. The second step consists in using the annihilator `ann_c` of the summand to reduce the resulting expression under the stairs of the Gröbner basis. In fact, this latter expression features several collections of terms, that will be reduced to as many independent copies of the stairs. In the present example, we observe two such collections: *(i)* terms that are around the lower bound $(n, 0)$ of the sum, of the form $c_{n,0}, \dots c_{n,s}$; *(ii)* terms that are around the upper bound (n, n) of the summation, of the form $c_{n,n}, \dots, c_{n,n+s}$ for a constant s. The border terms induce two such collections but there might be more, depending in particular on the shape of the `precond.CT_premise` proviso. For example, the sum $\sum_{k=0}^{n}(-1)^k\binom{n}{k}\binom{3k}{n} = (-3)^n$ leads to a proviso involving $n = 3k + 1$ and similar terms: an additional category of terms around $(n, n/3)$ drifts away from both $(n, 0)$ and (n, n) when n grows.

```
============================
P_cf2 n * c (n + 2) n + P_cf1 n * c (n + 1) n +
P_cf0 n * c n n + Q_cf0_0 n n * c n n + P_cf1 n * c (n + 1) (n + 1) +
P_cf2 n * c (n + 2) (n + 1) + P_cf2 n * c (n + 2) (n + 2) = 0
```

Listing 8.7: Terms around the upper bound

The collection of terms around the upper bound in our running example is displayed on Listing 8.7. The script of Listing 8.8 reduces this collection under the stairs of `ann_c`, producing the expression displayed on Listing 8.9. The premise of each rule in this basis being an integer linear arithmetic expression, we check its satisfiability using our front-end `intlia` to the `lia` proof command [13], which automates the formal proof of first-order formulae of linear arithmetics.

```
rewrite (ann_c.Sk (n + 2) (n + 1)); last by intlia.
rewrite (ann_c.Sk (n + 2) n); last by intlia.
rewrite (ann_c.Sk (n + 1) n); last by intlia.
rewrite (ann_c.Sn (n + 1) n); last by intlia.
rewrite (ann_c.Sn n n); last by intlia.
set cnn := c n n.
Fail set no_more_c := c _ _.
```

Listing 8.8: Reduction modulo the Gröbner basis of c

```
============================
P_cf2 n * Sn_cf0_0 (n + 1) n * Sn_cf0_0 n n * cnn + P_cf1 n * Sn_cf0_0 n n * cnn
     +
P_cf0 n * cnn + Q_cf0_0 n n * cnn + P_cf1 n * Sk_cf0_0 (n + 1) n * Sn_cf0_0 n n *
      cnn +
P_cf2 n * Sk_cf0_0 (n + 2) n * Sn_cf0_0 (n + 1) n * Sn_cf0_0 n n * cnn +
P_cf2 n * Sk_cf0_0 (n + 2) (n + 1) * Sk_cf0_0 (n + 2) n *
Sn_cf0_0 (n + 1) n * Sn_cf0_0 n n * cnn = 0
```

Listing 8.9: Rational function with folded coefficients

The third and last step consists in checking that the rational-function coefficient of every remaining evaluation of c is zero. For this purpose, we start by unfolding the definitions of the coefficients `P_cf2`, `Sn_cf0`. Previous steps kept them carefully folded as these values play no role in the previous normalizations but can lead to large expressions if expanded, significantly slowing down any other proof command. The resulting rational function is proved to be zero by a combination of the `field` [38] and `lia` [13] proof commands. The former reduces the rational equation into a polynomial one between the cross product of two rational functions. This equation is then solved by the `ring` proof command [85]. The algebraic manipulations performed by `field` produce a set of non-nullity conditions for the denominators. These are solved by the `lia` proof command. To this end, our Maple script generates rational fractions with factored denominators, that happen to feature only linear factors in these examples.

8.1.7 Composing closures and reducing the order of B

Figure 8.2 describes the global dependencies of the files proving all the steps in Table 8.1. In order to complete the formal proof of Lemma 13, we verify

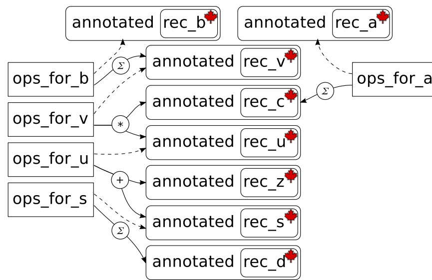


Figure 8.2: Formal proofs of Table 8.1

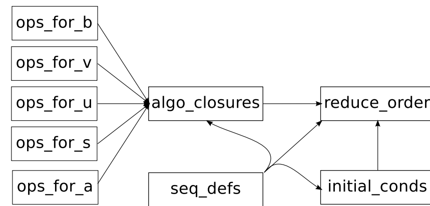


Figure 8.3: Formal proof of Lemma 13

formally in file `algo_closures` that each sequence involved in the construction of a_n and b_n is a solution of the corresponding Gröbner system of annotated recurrence, starting from c_n, d_n, and z_n and applying the lemmas proved in the `ops_for_*` files all the way to the the final conclusions of `ops_for_a` and `ops_for_b`. This proves that a_n is a solution of the recurrence (6.2) but provides only a recurrence of order four for b_n; let us call this recurrence R_4.

In file `reduce_order`, we prove that b as well satisfies the recurrence (6.2) using four evaluations b_0, b_1, b_2, b_3 that we compute in file `initial_conds`. The reasoning is the following: define b' as the unique sequence obtained by fixing $b'_0 = b_0$, $b'_1 = b_1$ and such that b' satisfies the Apéry recurrence (6.2) of order 2. Then prove that b' coincides with b for $n = 0, 1, 2, 3$. Finally, establish that any solution of Apéry's recurrence is also a solution of R_4. Thus $b' = b$, and this proves that b satisfies Apéry's recurrence.

8.2 Consequences of Apéry's recurrence

In this section, we detail the elementary proofs of the properties obtained as corollaries of Lemma 13. We recall from Chapter 6 that these properties describe

the asymptotic behavior of the sequence $\delta_n = a_n\zeta(3) - b_n$, with:

$$a_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2, \qquad b_n = a_n z_n + \sum_{k=1}^{n}\sum_{m=1}^{k} \frac{(-1)^{m+1}\binom{n}{k}^2\binom{n+k}{k}^2}{2m^3\binom{n}{m}\binom{n+m}{m}} \tag{8.14}$$

In this section, we use the vocabulary and notations of Cauchy real numbers (see Section 7.3). For instance, we have the equality between the two Cauchy reals:

Lemma 15. $\zeta(3) = \frac{b}{a}$

Proof. Easy from the definition of the sequences a and b. □

However, the Mathematical Components libraries do not cover any topic of analysis, and even the most basic definitions of transcendental functions like the exponential or the logarithm are not available. Fortunately, it is possible to obtain the required properties of the sequence δ by very elementary means, and almost all these elementary proofs can be inferred from a careful reading and a combination of Salvy's proof [111] and of van der Poorten's description [123].

Following van der Poorten, we introduce an auxiliary sequence $(w_n) \in \mathbb{Q}^n$, defined as:

$$w_n = \begin{vmatrix} b_{n+1} & a_{n+1} \\ b_n & a_n \end{vmatrix} = b_{n+1}a_n - a_{n+1}b_n.$$

The sequence w is called a Casoratian: as a and b are solutions of a same linear recurrence relation (6.2) of order 2, this can be seen as a discrete analogue of the Wronskian for linear differential systems. For example, w satisfies a recurrence relation of order 1, which provides a closed form for w:

Lemma 16. *For $n \geq 2$, $w_n = \frac{6}{(n+1)^3}$.*

Proof. Since a and b satisfy the recurrence relation (6.2), we have for $n \in \mathbb{N}$:

$$\begin{pmatrix} b_{n+2} & a_{n+2} \\ b_{n+1} & a_{n+1} \end{pmatrix} = \begin{pmatrix} \frac{(17n^2+51n+39)(2n+3)}{(n+2)^2} & -\frac{(n+1)^3}{(n+2)^3} \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} b_{n+1} & a_{n+1} \\ b_n & a_n \end{pmatrix}.$$

Taking determinants on both sides, we get

$$w_{n+1} = \frac{(n+1)^3}{(n+2)^3} w_n.$$

The result then follows from the computation of w_0. □

From this formula, we can obtain the positivity of the sequence δ, and an evaluation of its asymptotic behavior in terms of the sequence a.

Corollary 1. *For any $n \in \mathbb{N}, 0 < \zeta(3) - \frac{b_n}{a_n}$.*

Proof. Since $\zeta(3) = \frac{b}{a}$, it is sufficient to show that for any $k < l$, we have $0 < \frac{b_l}{a_l} - \frac{b_k}{a_k}$. Thus it is sufficient to observe that for any k, we have $0 < \frac{b_{k+1}}{a_{k+1}} - \frac{b_k}{a_k}$, which follows from Lemma 16. □

Corollary 2. $\zeta(3) - \frac{b_n}{a_n} = \mathcal{O}(\frac{1}{a_n^2})$.

Proof. Since $\zeta(3) = \frac{b}{a}$, it is sufficient to show that there exists a constant K, such that for any $k < l$, $\frac{b_l}{a_l} - \frac{b_k}{a_k} \leq \frac{K}{a_k^2}$. But since a is an increasing sequence, Lemma 16 proves that for any $k < l$, $\frac{b_l}{a_l} - \frac{b_k}{a_k} \leq \sum_{i=k}^{l-1} \frac{w_i}{a_i a_{i+1}} \leq \sum_{i=k}^{l-1} \frac{6}{(i+1)^3 a_k^2} \leq \frac{K}{a_k^2}$, for any K greater than $6\zeta(3)$. □

The last remaining step of the proof is to show that the sequence a grows fast enough.

Lemma 17. $33^n = \mathcal{O}(a_n)$.

Proof. Consider the auxiliary sequence $\rho_n = \frac{a_{n+1}}{a_n}$. Since ρ_{51} is greater than 33, we only need to show that the sequence ρ is increasing. For the sake of readability, we denote μ_n and ν_n the fractions coefficients of the recurrence satisfied by a, obtained from Equation (6.2) after division by its leading coefficient. Thus a satisfies the recurrence relation:

$$a_{n+2} - \mu_n a_{n+1} + \nu_n = 0.$$

For $n \in \mathbb{N}$, we also introduce the function $h_n(x) = \mu_n + \frac{\nu_n}{x}$, so that $\rho_{n+1} = h_n(\rho_n)$. The polynomial $P_n(x) = x^2 - \mu_n x + \nu_n$ has two distinct roots $x'_n < x_n$, and the formula describing the roots of polynomials of degree 2 show that $0 < x'_n < 1 < x_n$ and that the sequence x_n is increasing. But since $h_n(x) - x = -\frac{P_n(x)}{x}$, for $1 < x < x_n$, we have $h_n(x) > x$. A direct recurrence shows that for any $n \geq 2$, $\rho_n \in [1, x_n]$, which concludes the proof. □ □

In the formal proof of Lemma 17, the computation of ρ_{51} was made possible by using the CoqEAL library, with the help of Cyril Cohen. This proof also requires a few symbolic computations that are a bit tedious to perform by hand: in these cases, we used Maple as an oracle to massage algebraic expressions, before formally proving the correctness of the simplification. This was especially useful to study the roots x'_n and x_n of P_n.

We can now conclude with the limit of the sequence $\ell_n^3 \delta_n$, under the assumption that $\ell_n = \mathcal{O}(3^n)$.

Corollary 3. $\lim\limits_{n \to \infty} (\ell_n^3 \delta_n) = 0$.

Proof. Immediate, since $\delta_n = \mathcal{O}(\frac{1}{a_n})$ by Corollary 2, and $\ell_n^3 = \mathcal{O}((3^3)^n)$, and $3^3 < 33$. □

In the next Section, we describe the proof of the last remaining assumption, about the asymptotic behavior of ℓ_n.

8.3 Asymptotics of $lcm(1, ..., n)$

The asymptotic behavior of the sequence $lcm(1, ..., n)$ is an elementary corollary of the Prime Number Theorem. The proof of this corollary is based on a simple remark about the p-valuations of $lcm(1, ..., n)$.

Remark 2. *For any integer n, let ℓ_n denote the least common multiple of the integers no greater than n, denoted $lcm(1, ..., n)$. For any prime number p, $p^{v_p(\ell_n)}$ is the highest power of p not exceeding n, so that:*

$$v_p(\ell_n) = \left\lfloor \log_p(n) \right\rfloor.$$

Proof. Noticing that $v_p(lcm(a,b)) = \max(v_p(a), v_p(b))$, we see by induction on n that $v_p(\ell_n) = \max_{i=1}^{n} v_p(i)$. Recall from Section 7.1 that $\lfloor \log_p(n) \rfloor$ is a notation for the greatest integer α such that $p^\alpha \leq n$. Since $\alpha = v_p(p^\alpha)$, we have $\alpha \leq v_p(\ell_n)$. Now suppose that $v_p(\ell_n) = v_p(i)$ for some $i \in \{1, \cdots, n\}$. Then $i = p^{v_p(i)} q$ with $gcd(p,q) = 1$ so that $p^{v_p(\ell_n)} = p^{v_p(i)} \leq i \leq n$ and thus $v_p(\ell_n) \leq \alpha$. This proves that $v_p(\ell_n) = \alpha$.

□

By Remark 2, ℓ_n can hence be written as $\prod_{p \leq n} p^{\lfloor \log_p(n) \rfloor}$ and therefore:

$$\ln(\ell_n) = \sum_{p \leq n} \lfloor \log_p(n) \rfloor \ln(p).$$

If $\pi(n)$ is the number of prime numbers no greater than n, we hence have:

$$\ln(\ell_n) \sim \pi(n) \ln(n).$$

The Prime Number theorem states that $\pi(n) \sim \frac{n}{\ln(n)}$; we can thus conclude:

$$\ell_n = e^{n\,(1+o(1))}.$$

J. Avigad and his co-authors provided the first machine-checked proof of the Prime Number theorem [6], which was considered at the time as a formalization *tour de force*. Their formalization is based on a proof by A. Selberg and P. Erdös. Although the standard proofs of this theorem use tools from complex analysis like contour integrals, their choice was guided by the corpus of formalized mathematics available for the Isabelle proof assistant, or the limits thereof. Although less direct, the proof by A. Selberg and P. Erdös is indeed more elementary and avoids complex analysis completely.

8.3.1 Notations and Outline

In order to prove Corollary 3 in Section 8.2, we already mentioned in Chapter 6 that a weaker description of the asymptotic behavior of (ℓ_n) is sufficient, stated in Lemma 11:

$$\ell_n = \mathcal{O}(3^n). \tag{8.15}$$

This sort of weaker description was in fact known before the first proofs of the Prime Number theorem but our formal proof is a variation on an elementary proof proposed by Hanson [62].

The idea of the proof is to replace the study of ℓ_n by that of another sequence $C(n)$. The latter is defined as a multinomial coefficient depending on elements of a fast-growing sequence α. The fact that $\prod_{i=1}^{n} \alpha_i^{1/\alpha_i} < 3$ independently of n then allows to show that $C(n) = \mathcal{O}(3^n)$.

8.3.2 Proof

Define the sequence $(\alpha_n)_{n \in \mathbb{N}}$ by $\alpha_1 = 2$, 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. Hanson directly works with $C(n)$, but we found this to be inconvenient to manipulate in the proof. Moreover, most inequalities stated on $C(n)$ actually hold for $C(n,k)$ with little or no more hypotheses. Notice that $C(n,k) = \binom{n}{\lfloor n/\alpha_1\rfloor, \lfloor n/\alpha_2\rfloor, \cdots, \lfloor n/\alpha_k\rfloor} \cdot \left(n - \sum_{i=0}^{k} \left\lfloor \frac{n}{\alpha_i} \right\rfloor\right)!$. In particular, $C(n,k) \in \mathbb{N}$. The goal is now to show that $\ell_n \leq C(n) < K \cdot 3^n$ for some K.

Lemma 18. *For $k \in \mathbb{N}$,*

$$\sum_{i=1}^{k} \frac{1}{\alpha_i} = \frac{\alpha_{k+1} - 2}{\alpha_{k+1} - 1} < 1 \text{ and thus for } x \in \mathbb{Q} \text{ with } x \geq 1, \lfloor x \rfloor > \sum_{i=1}^{k} \left\lfloor \frac{x}{\alpha_i} \right\rfloor.$$

The proof is done by induction and relies on the fact that if $a \in \mathbb{Q}$ and $m \in \mathbb{N}^+$, we have $\lfloor \frac{a}{m} \rfloor = \left\lfloor \frac{\lfloor a \rfloor}{m} \right\rfloor$.

In the following, for $n, k \in \mathbb{N}$ and p prime, we denote $\beta_p(n,k)$ for $val_p(C(n,k))$.

Lemma 19. *For all $n, k \in \mathbb{N}$ and p prime, $\beta_p(n,k) \geq \lfloor \log_p(n) \rfloor = val_p(\ell_n)$. Therefore $C(n,k) \geq \ell_n$.*

Proof. The proof uses Lemma 14.

$$\begin{aligned}
\beta_p(n,k) =& val_p(n!) - \sum_{i=1}^{k} val_p(\lfloor n/\alpha_i \rfloor !) \\
=& \sum_{i=1}^{\lfloor \log_p(n) \rfloor} \lfloor n/p^i \rfloor - \sum_{i=1}^{k} \sum_{j=1}^{\lfloor \log_p(\lfloor \frac{n}{\alpha_i} \rfloor) \rfloor} \left\lfloor \frac{n}{\alpha_i p^j} \right\rfloor \\
=& \sum_{i=1}^{\lfloor \log_p(n) \rfloor} \left(\lfloor n/p^i \rfloor - \sum_{j=1}^{k} \left\lfloor \frac{\lfloor \frac{n}{p^i} \rfloor}{\alpha_j} \right\rfloor \right) \\
\geq& \sum_{i=1}^{\lfloor \log_p(n) \rfloor} 1 \text{ (thanks to the fact that } \sum \frac{1}{\alpha_i} < 1 \text{ from lemma 18).}
\end{aligned}$$

Since $\ell_n = \prod_{p \leq n} p^{\lfloor \log_p(n) \rfloor}$ from remark 2, we get $\ell_n \leq C(n,k) = \prod_{p \leq n} p^{\beta_p(n,k)}$. □

Lemma 20. *For $i \geq 1$ and $n \geq \alpha_i$,*

$$\frac{\left(\frac{n}{\alpha_i}\right)^{\frac{n}{\alpha_i}}}{\left\lfloor \frac{n}{\alpha_i} \right\rfloor^{\lfloor \frac{n}{\alpha_i} \rfloor}} < \left(\frac{10\, n}{\alpha_i}\right)^{\frac{\alpha_i - 1}{\alpha_i}}.$$

Proof. If $n = \alpha_i$, we have $1 < \sqrt{10} \leq 10^{\frac{\alpha_i - 1}{\alpha_i}}$, hence the result. Otherwise $n > \alpha_i$ and then, if we write $n = b\alpha_i + r$, with $0 \leq r < \alpha_i$, we have $\frac{n - \alpha_i + 1}{\alpha_i} - \left\lfloor \frac{n}{\alpha_i} \right\rfloor = -1 + \frac{1}{\alpha_i} < \frac{1}{2} < 0$ so that $\frac{n - \alpha_i + 1}{\alpha_i} < \left\lfloor \frac{n}{\alpha_i} \right\rfloor$.

It is easy to deduce that $\left(\frac{n-\alpha_i+1}{\alpha_i}\right)^{\frac{n-\alpha_i+1}{\alpha_i}} < \left\lfloor\frac{n}{\alpha_i}\right\rfloor^{\lfloor\frac{n}{\alpha_i}\rfloor}$, hence we have

$$\frac{\left(\frac{n}{\alpha_i}\right)^{\frac{n}{\alpha_i}}}{\left\lfloor\frac{n}{\alpha_i}\right\rfloor^{\lfloor\frac{n}{\alpha_i}\rfloor}} < \frac{\left(\frac{n}{\alpha_i}\right)^{\frac{n}{\alpha_i}}}{\left(\frac{n-\alpha_i+1}{\alpha_i}\right)^{\frac{n-\alpha_i+1}{\alpha_i}}} = \left(1+\frac{\alpha_i-1}{n-\alpha_i+1}\right)^{\frac{n-\alpha_i+1}{\alpha_i-1}\frac{\alpha_i-1}{\alpha_i}} \cdot \left(\frac{n}{\alpha_i}\right)^{\frac{\alpha_i-1}{\alpha_i}}.$$

The first operand in the last expression is of the shape $\left(\left(1+\frac{1}{x}\right)^x\right)^{\frac{\alpha_i-1}{\alpha_i}}$, where x is rational. We showed using only elementary properties of algebraic numbers that for $x \in \mathbb{Q}$, $0 < x$, $\left(1+\frac{1}{x}\right)^x < 10$, hence the result. Note that we only needed that there exist a constant $K > 0$ such that $\left(1+\frac{1}{x}\right)^x < K$. □

Of course, using elementary real analysis allows for the tighter bound e for $\left(1+\frac{1}{x}\right)^x$, which was used in Hanson's paper, but this bound is irrelevant for the final result. Using the previous facts, we obtain the following bound:

Lemma 21. *For $k \geq 1$ and $n \geq 2$,*

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

Proof. First observe that if $m = m_1 + \cdots + m_k$ where m and the m_i are (not all zero) nonnegative integers, we have because of Theorem 1:

$$(m_1 + \cdots + m_k)^m \geq \binom{m}{m_1, \cdots, m_k} m_1^{m_1} \cdots m_k^{m_k}. \tag{8.16}$$

Let $k \geq 1$, and define:

$$t = \sum_{i=1}^{k} \left\lfloor \frac{n}{\alpha_i} \right\rfloor.$$

Then $t < n$ by lemma 18. We have

$$C(n,k) = n \cdot (n-1) \cdots (t+1) \binom{t}{\lfloor n/\alpha_1\rfloor, \lfloor n/\alpha_2\rfloor, \cdots, \lfloor n/\alpha_k\rfloor}. \tag{8.17}$$

Because of equation (8.16), we know that

$$\binom{t}{\lfloor n/\alpha_1\rfloor, \lfloor n/\alpha_2\rfloor, \cdots, \lfloor n/\alpha_k\rfloor} \leq \frac{t^t}{\lfloor n/\alpha_1\rfloor^{\lfloor n/\alpha_1\rfloor}\lfloor n/\alpha_2\rfloor^{\lfloor n/\alpha_2\rfloor}\cdots\lfloor n/\alpha_k\rfloor^{\lfloor n/\alpha_k\rfloor}}. \tag{8.18}$$

From equations (8.17) and (8.18) we deduce that

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

□

Lemma 22. *Let $k \geq 3$, $n \in \mathbb{N}$. If $\alpha_k \leq n$ then*

$$k < \lfloor \log_2 \lfloor \log_2 n \rfloor \rfloor + 2.$$

Proof. First observe by a simple induction that for all $k \geq 3$, $\alpha_k > 2^{2^{k-2}} + 1$ so that $k - 2 < \lfloor \log_2 (\lfloor \log_2 \alpha_k \rfloor) \rfloor \leq \lfloor \log_2 (\lfloor \log_2 n \rfloor) \rfloor$. ☐

Lemma 23. *Let $k \geq 1$, $n \in \mathbb{N}$. If $\alpha_k \leq n$,*

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

Proof. The result is straightforward by combining lemmas 20 and 21. ☐

Lemma 24. *Let $w_k = \prod_{i=1}^{k} \alpha_i^{\frac{1}{\alpha_i}}$, $k \geq 1$. Then w_k is increasing and there exists $w \in \mathbb{R}$, with*

$$w < 2.98,$$

such that $w_k < w$.

Proof. The sequence w_k is increasing because $\alpha_i^{\frac{1}{\alpha_i}} > 1$ (because $\alpha_i > 1$). Since $\alpha_i^2 > \alpha_{i+1} > (\alpha_i - 1)^2$, one can see that for $i \geq 3$, $\alpha_{i+1}^{\frac{1}{\alpha_{i+1}}} < \sqrt{\alpha_i^{\frac{1}{\alpha_i}}}$, so that for all $k \geq 1$ and $l \geq 0$,

$$w_{k+l} \leq \prod_{i=1}^{k} \alpha_i^{\frac{1}{\alpha_i}} \cdot \alpha_{k+1}^{\frac{1}{\alpha_{k+1}} \sum_{i=0}^{l} \frac{1}{2^i}} \leq w_k \cdot \alpha_{k+1}^{\frac{2}{\alpha_{k+1}}}.$$

We establish by an elementary external computation verified in Coq that $\alpha_1^{\frac{1}{\alpha_1}} < \frac{283}{200}$, $\alpha_2^{\frac{1}{\alpha_2}} < \frac{1443}{1000}$, $\alpha_3^{\frac{1}{\alpha_3}} < \frac{1321}{1000}$, $\alpha_4^{\frac{1}{\alpha_4}} < \frac{273}{250}$ and $\alpha_5^{\frac{1}{\alpha_5}} < \frac{201}{200}$. From the bound above with $k = 4$ we get $w < w_4 \cdot \alpha_5^{\frac{2}{\alpha_5}} \leq \frac{5949909309448377}{2 \cdot 10^{15}} < 2.98$. ☐

Remark 3. *For $k \geq 1$, we have*

$$\frac{\alpha_1 - 1}{\alpha_1} + \frac{\alpha_2 - 1}{\alpha_2} + \cdots + \frac{\alpha_k - 1}{\alpha_k} = k - 1 + \frac{1}{\alpha_{k+1} - 1}.$$

Proof. It is a direct consequence of Lemma 18. ☐

Note that the statement of Remark 3 actually corrects a typo in the original paper.

Theorem 6. *If $\alpha_k \leq n < \alpha_{k+1}$,*

$$C(n,k) = C(n) < (10\,n)^{k - \frac{1}{2}} w^{n+1}.$$

Proof. From lemma 23, recall that we have

$$\begin{aligned} C(n,k) <& \frac{n^n \left(\frac{10\,n}{\alpha_1}\right)^{\frac{\alpha_1-1}{\alpha_1}} \left(\frac{10\,n}{\alpha_2}\right)^{\frac{\alpha_2-1}{\alpha_2}} \cdots \left(\frac{10\,n}{\alpha_k}\right)^{\frac{\alpha_k-1}{\alpha_k}}}{\left(\frac{n}{\alpha_1}\right)^{\frac{n}{\alpha_1}} \left(\frac{n}{\alpha_2}\right)^{\frac{n}{\alpha_2}} \cdots \left(\frac{n}{\alpha_k}\right)^{\frac{n}{\alpha_k}}} \\ =& \frac{n^n (10\,n)^{\left(\sum\limits_{i=1}^{k} \frac{\alpha_i-1}{\alpha_i}\right)} \left(\prod_{i=1}^k \alpha_i^{\frac{1}{\alpha_i}}\right)^n}{n^{n\sum_{i=1}^k \frac{1}{\alpha_i}} \prod_{i=1}^k \alpha_i^{\frac{\alpha_i-1}{\alpha_i}}}. \end{aligned}$$

It can be seen using lemma 18 that:

$$n^{n\left(1-\sum_{i=1}^k \frac{1}{\alpha_i}\right)} \leq n.$$

Thus

$$\begin{aligned} C(n,k) <& n\frac{(10\,n)^{k-1+\frac{1}{\alpha_{k+1}-1}} w_k^n}{\prod_{i=1}^k \alpha_i^{\frac{\alpha_i-1}{\alpha_i}}} \text{ thanks to remark 3} \\ \leq& n\frac{(10\,n)^{k-1+\frac{1}{\alpha_{k+1}-1}} w^n}{\prod_{i=1}^k \alpha_i^{\frac{\alpha_i-1}{\alpha_i}}} \text{ because } w_k \leq w. \end{aligned}$$

Since $n < \alpha_{k+1} = 1 + \prod_{i=1}^k \alpha_i$, $n \leq \prod_{i=1}^k \alpha_i$, and we have

$$\prod_{i=1}^{k} \alpha_i^{\frac{\alpha_i-1}{\alpha_i}} = \frac{\prod\limits_{i=1}^{k} \alpha_i}{w_k} \geq \frac{n}{w_k}.$$

Thus

$$\begin{aligned} C(n,k) <& (10\,n)^{k-1+\frac{1}{\alpha_{k+1}-1}} w^n w_k \\ \leq& (10\,n)^{k-\frac{1}{2}} w^{n+1} \text{ as } \alpha_{k+1} \geq 3 \text{ and } w_k \leq w. \end{aligned}$$

□

We can now prove lemma 11:

Proof.

$$\ell_n/3^n \leq C(n,k)/3^n = (10\,n)^{k-\frac{1}{2}} \left(\frac{w}{3}\right)^{n+1}.$$

Remembering that $k < \lfloor \log_2 \lfloor \log_2 n \rfloor \rfloor + 2$ and $w < 3$, it is elementary to show that the quantity on the right is eventually decreasing to 0 and therefore bounded, which proves the result. We once again make use of the fact that $\left(1+\frac{1}{x}\right)^x$ is bounded in the course of this elementary proof. □

Chapter 9

Conclusion

9.1 Proving Inequalities in Coq

In the proof of Section [62], we repeatedly had to prove inequalities between expressions in an ordered ring or field. This was quite tedious for two reasons. We present these reasons and suggest ideas for tools which could relieve the user. Such tools would replace something humans do without thinking about it.

Chained Inequalities: Proofs of statements of the style $e_1 \leq e_n$ which are done using a chain of inequalities $e_1 \leq e_2 \leq \cdots \leq e_n$ are currently wearisome, especially in the case where the e_i are big expressions. The way one typically proceeds is by first proving that $e_1 \leq e_2$, which involves writing down the expressions of e_1 and e_2 in Coq. Once this is done, one invokes transitivity of inequality so that one is left with the goal $e_2 \leq e_n$. One then states $e_2 \leq e_3$, which again involves writing down both e_2 and e_3, and so on. This incurs a lot of manual copy-pasting, which makes the code inelegant and the work of doing proofs tedious. One could hope for something similar as the "calc" environment in the Lean theorem prover [37]. For example, we could have a tool allowing something in the style

```
Goal e_1 <= e_n.
- # <= e_2.
  by proof_1.
...
- # <= e_{n-1}.
  by proof_{n-1}
(* the final goal is e_{n-1} <= e_n *)
- by proof_n.
```

where the vernacular `# <= X` on a goal of the shape $a \leq b$ would apply transitivity and create the two new goals $a \leq X$ and $X \leq b$.

Side Conditions: We lack a tool for the automation of side-condition proving in inequalities. Suppose we want to prove inequalities in an ordered ring R between products $x_1 \cdot y_1 \cdot z_1 \leq a_1 \cdot b_1 \cdot c_1$, where x_1, z_1, a_1 and c_1 are a (possibly empty) product of nonnegative expressions, and y_1 and b_1 are nonnegative

elements of R.

Typically, this is done by inlining inequalities on y_1 and b_1 alone. For example, we may know that $y_1 \leq y_2$, and we can use this fact so that the initial goal $x_1 \cdot \mathbf{y_1} \cdot z_1 \leq a_1 \cdot b_1 \cdot c_1$ becomes $x_1 \cdot \mathbf{y_2} \cdot z_1 \leq a_1 \cdot b_1 \cdot c_1$.

Thus, we do a series of 'rewritings' of the shape $x_1 \cdot \mathbf{y_1} \cdot z_1 \leq a_1 \cdot b_1 \cdot c_1 \rightsquigarrow x_1 \cdot \mathbf{y_2} \cdot z_1 \leq a_1 \cdot b_1 \cdot c_1$, justified by proving that $y_1 \leq y_2$. To do so, the only lemma at our disposal is typically of the shape:

$$\forall a, b, c, d, 0 \leq a \Rightarrow 0 \leq b \Rightarrow a \leq b \Rightarrow c \leq d \Rightarrow a \cdot c \leq b \cdot d.$$

This means that we will have to re-prove several times that each quantity appearing in the product is nonnegative in the course of the rewritings. Moreover, this lemma only works directly for products of *two* elements of R; when we apply it to the product of k elements, we need to apply it $k-1$ times. .

A tool to tackle this problem would have to be able to deal with products of arbitrary arity, to propose a sensible deterministic mechanism to describe *where* and *how* we want to rewrite inequalities, and to keep track of already proven goals (typically the non-negativity of terms in the product).

9.2 Casts

As mentioned in Section 7.2, inclusions between sets in the Mathematical Components hierarchy are represented by injective functions between datatypes. Sometimes, coercions and notations can make two different representations indistinguishable to the eye in a Coq session. One consequence is that the user will fail to be able to rewrite or apply a theorem on what seems to be a valid instance. For example, if `n` is of type `nat`, `Posz (n + 1)` and `(Posz n) + 1` can both appear as `n+1`, because the coercion `Posz` is hidden by default. Rewriting `n + 1` to `S n` (successor of `n`) will succeed in the first case and fail in the second.

For a similar reason, proof commands like `ring` will fail on what seems like a valid equation, and only careful inspection of the Coq terms in the goal will reveal the source of failure. In fact, we had to rewrite tactics which started by normalizing expressions to a standard form in order to be able to apply the `ring` and `field` proof commands to the type `rat` representing the set $\mathbb{Q}$.

However, the formalization of nontrivial mathematics would not be possible if it were not for the abundant notations and coercions at our disposal to be able to "forget" the precise nature of every object manipulated, just like we do when we write proofs on paper. A balance must be struck between an abundance of tools which may lead to not understanding what object we are manipulating, or which may lead to the inexplicable failure of proof automation tools, and a lack of such tools which would prevent formal proofs from being done.

9.3 Related Work

Number theory is an area of mathematics where very simple statements often require extremely difficult proofs. We are not aware of a comprehensive formal proof library on the topic, although there exist formal proofs of a few emblematic results. The elementary fact that $\sqrt{2}$ is irrational was used as an example problem in a comparative study of the styles of various theorem provers [126],

including Coq. The Prime Number theorem was proved formally for the first time by Avigad et al. [6] and later by Harrison [63]. The transcendence of e was formalized by Bingham [16] in HOL-Light. More recently, the transcendence of both π and e was formalized in Coq [11]. Some of the ingredients needed in the present proof are however not specific to number theory. For instance, we here use a very basic infrastructure to represent asymptotic behaviors, but "big Oh" notations have been discussed by Avigad [6] in his formalization of the Prime Number Theorem, and by Boldo et al. [19] in a continuous context. Another example of such a secondary topic is the theory of multinomial coefficients, which is also relevant to combinatorics, and which is also defined by Hivert in his Coq library Coq-Combi [73]. However, up to our knowledge this library does not feature a proof of the generalized Newton identity.

We used Cohen's Cauchy reals to define $\zeta(3)$, and algebraic complex numbers in our simplification of Hanson's proof. We considered using instead the C-CoRN library of constructive analysis [35], but we found out that it would have been too much work compared to what we eventually did, in part because of the necessity to write a "bridge" to link mathematical structures from the Mathematical Components libraries to those of C-CoRN.

Harrison [64] recently presented a way to produce rigorous proofs from certificates produced by the Wilf-Zeilberger certificates, by seeing sequences as limits of complex functions. His method applies to the sequence a, which satisfies the recurrence Equation (6.2). However, this method does not allow for a proof that b satisfies Equation (6.2), because the summand is itself a sum but is not hypergeometric. Up to our knowledge, there is no way to justify the output of the efficient algorithms of creative telescoping used here without handling a trace of provisos.

As already mentioned, the idea to use computer algebra software (CAS) as an oracle outputting a certificate to be checked by a theorem prover, dubbed a *skeptic's* approach, was described in several papers [8] [7] [65]. This technique takes the best of both worlds to produce reliable proofs requiring large scale computations. In the case of Coq, this viewpoint is especially fruitful since the kernel of the proof assistant includes efficient evaluation mechanisms for the functional programs written inside the logic [57]. Notable successes based on this idea include the use of Pocklington certificates to check primality inside Coq [58] or external computations of commutative Groebner bases, with applications for instance in geometry [106]. Delahaye and Mayero proposed [38] to use CAS to help experimenting with algebraic expressions inside a proof assistant, before deciding what to prove and how to prove it. Unfortunately, their tool was not usable in our case, where algebraic expressions are made with operations that come from a hierarchy of structures.

Organizing the cooperation of a CAS and a proof assistant sheds light on their respective differences and drawbacks. The initial motivation of this work was to study the algorithms used for the automatic discovery and proof of recurrences. Our hope was to be able to craft an automated tool providing formal proofs of recurrences, by using the output of these algorithms, in a skeptical way. This plan did not work and Section 8.1 illustrates the impact of confusing the rational fractions manipulated by symbolic computations with their evaluations, which should be guarded by conditions on the denominators. On the other hand, proof assistants need to be handled carefully if one wants to manipulate the large expressions imported during the cooperation, even those which are of

a small to moderate size for the standards of computer algebra systems. For instance, we have highlighted in our previous report [26] the necessity to combine two distinct natures of data-structures in our libraries: one devoted to formal proofs, which may use computation inside the logic to ease bureaucratic steps in formal proofs, and one devoted to large scale computations, which provides a fine-grained control on the complexity of operations.

On several occasions in this work, we wrote more elementary versions of the proofs than what we had found in the texts we were formalizing. We agree with Avigad [6] when he says that this can be both frustrating and enjoyable: on one hand, it can illustrate the lack of mathematical libraries for theorems which mathematicians would find simple, such as elementary analysis for studying the asymptotics of sequences as in Section 8.2. Ten years later, "the need for elementary workarounds" is still present, despite his fear that it would "gradually fade, and with it, alas, one good reason for investing time in such exercises"[6]. On the other hand, this need gives an opportunity to better understand the minimal scope of mathematical theories used in a proof, with the help of a computer. For instance, it was not clear to us that we could manage to completely avoid the need to define transcendental functions, or even the constant e, to formalize Hanson's paper [62], although it sometimes comes at the price of some slightly pedestrian calculations.

Part III

Certified Computations

Chapter 10

Introduction

Sometimes validating external computations is not an option. For some problems, there does not seem to exist a certificate for which verification is cheaper than computation. In those cases, what matters is that there exist a *formal specification* of the way the result is obtained, so that one can prove that this result is correct *by construction*.

The problem described in this part of the thesis falls in the latter category. The goal is to estimate integrals numerically: given two bounds u and v and a function $f : \mathbb{R} \to \mathbb{R}$, find A and B such that

$$A \leq \int_u^v f(t)dt \leq B \tag{10.1}$$

and with the interval $[A; B]$ preferably tight. This problem was already posed long before computers, for example when trying to estimate π by enclosing the unit disk between two n-gons. Numerical integration distinguishes itself from symbolic integration which aims at finding an equivalent exact mathematical formula for an integral, for example:

$$\int_0^1 \cos(t)dt = \sin(1).$$

However, this is not always possible using elementary functions, as in the example, for $x \in \mathbb{R}$, of

$$\int_0^x e^{t^2} dt.$$

Harnessing the power of computers to obtain a precise numerical value for previously unattainable integrals is becoming an issue for mathematicians, as estimates of numerous values of integrals become part of some modern mathematical proofs, such as that of the Ternary Goldbach Conjecture by Helfgott [70] or the Double Bubble Conjecture by Hass and Schlafly [66].

The problem of numerical integration is typically presented as an *approximation* problem. The goal is to find a natural number n, as well as n points $u \leq x_1 \leq x_2 \leq \cdots \leq x_n \leq v$ and n weights $w_1, w_2, \cdots, w_n \in \mathbb{R}$ such that that the integral of f is well approximated by a linear combination of evaluations of f:

$$\int_u^v f \simeq w_1 f(x_1) + w_2 f(x_2) + \cdots + w_n f(x_n). \tag{10.2}$$

Many numerical integration methods exist with different choices of weights. We borrow from Fousse [45] the following list: the family of Newton-Cotes methods, the Romberg method, the Gaussian quadrature methods, the Gauss-Kronrod quadrature, the Clenshaw-Curtis method and the double-exponential method. With enough hypotheses on f and its derivatives when they exist, each method comes with an error bound for the expression on the right of Equation (10.2).

As an example, the 2-point trapezoidal rule, which is an instance of the Newton-Cotes method, uses $n = 2$, $x_1 = u$, $x_2 = v$, and $w_1 = w_2 = \frac{v-u}{2}$. Then if $f \in \mathcal{C}^2([u; v])$,

$$\left| \int_u^v f(t) - \frac{v-u}{2} \left(f(x_1) + f(x_2) \right) \right| = -\frac{(v-u)^3 f''(\xi)}{12} \tag{10.3}$$

for some $\xi \in [u; v]$[1].

The first problem with this approach is that this error bound analysis is done assuming that all the performed operations (e.g. additions, multiplications, divisions) are exact. Most of the actual implementations of quadrature methods are made using floating-point number operations, and are thus prone to round-off errors.

Another problem is that an error bound such as in Equation (10.3) is often not used systematically by mathematical software to provide an error interval at the end of the computation, but simply as a heuristic to both justify that the method does eventually converge to the value of the integral, and to know how to adjust the parameters to reach the user-requested error.

There exists *rigorous* software which takes into account both these pitfalls. By "rigorous", we mean that the correctness of the method used by the software is proved on paper. The INTLAB [109] library by Rump uses a rigorous Romberg method [108], which is a Newton–Cotes quadrature formula, to integrate four times differentiable functions using interval arithmetic for both the value and the error bound. The software library Correctly Rounded Quadrature (CRQ) by Fousse [45] implements integration algorithms with rigorous error bounds for the Newton-Cotes and Gauss-Legendre methods.

At the end of his PhD thesis, Fousse calls for the validation of his algorithms using formal proofs. To our knowledge, formally verified computations currently provide the strongest assurance of correctness. This is appropriate in order to obtain the kind of guarantee required by large mathematical proofs. The main contribution of this part is the description of a tool to automatically verify integral enclosures inside the Coq proof assistant.

In Section 11, we start by giving an introduction to the efficient and rigorous numeric real computation methods which we use to obtain our enclosures, along with the Coq libraries we rely on.

[1] A proof of this error bound along with ones for more advanced quadrature methods are detailed in the book Methods of Numerical Integration [36] by Davis and Rabinowitz.

In Section 12, we describe how we obtain formally proved enclosures of both *proper* and *improper* integrals inside the Coq proof assistant. We present benchmarks on various integrals found in the literature. These benchmarks include two instances where we found previously undetected mistakes.

Finally, in Section 13, we compare these results with a custom implementation of Simpson's method; on the way, we present several iterations of our attempts to implement automatic differentiation of order n, for fun and profit.

Chapter 11

Efficient and Rigorous Numeric Real Computations

11.1 Interval Arithmetic

11.1.1 Notations and definitions

In the following, an *interval* is a closed connected subset of the set of real numbers. We use $\mathbb{I}$ to denote the set of intervals:

$$\mathbb{I} := \{[a; b] \mid a, b \in \mathbb{R} \cup \{\pm\infty\}\}.$$

A *point interval* is an interval of the shape $[a; a]$ where $a \in \mathbb{R}$. The set $\mathbb{I}$ can be equipped with an addition

$$\begin{aligned} +_{\mathbb{I}} :& \mathbb{I} \times \mathbb{I} \to \mathbb{I} \\ & ([a; b], [c; d]) \mapsto [a + c; b + d] \end{aligned}$$

with neutral element $[0; 0]$ which makes $\mathbb{I}$ a commutative monoid. The set $\mathbb{I}$ can also be equipped with a multiplication

$$\begin{aligned} \cdot_{\mathbb{I}} :& \mathbb{I} \times \mathbb{I} \to \mathbb{I} \\ & ([a; b], [c; d]) \mapsto [\min(a \cdot c, a \cdot d, b \cdot c, b \cdot d); \max(a \cdot c, a \cdot d, b \cdot c, b \cdot d)] \end{aligned}$$

with neutral element $[1; 1]$. When there will be no ambiguity, $+_{\mathbb{I}}$ (resp. $\cdot_{\mathbb{I}}$) will be simply noted $+$ (resp. $\cdot$).

Any interval variable will be denoted using a bold font. For any interval $\mathbf{x} \in \mathbb{I}$, $\inf \mathbf{x}$ (resp. $\sup \mathbf{x}$) denotes its left (resp. right) bound, with $\inf \mathbf{x} \in \mathbb{R} \cup \{-\infty\}$ (resp. $\sup \mathbf{x} \in \mathbb{R} \cup \{+\infty\}$). An *enclosure* of $x \in \mathbb{R}$ is an interval $\mathbf{x} \in \mathbb{I}$ such that $x \in \mathbf{x}$.

Definition 11. *The* closed convex hull *of a set $A \subseteq \mathbb{R}$ is the smallest interval containing A, denoted here* hull(A). *Moreover, the interval* hull$(\mathbf{a}, \mathbf{b})$ *denotes*

the convex hull of (the union of) two intervals $\mathbf{a}$ *and* $\mathbf{b}$. *Finally,* $\text{hull}(\mathbf{a}, +\infty)$ *designates the interval* $[\inf \mathbf{a}; +\infty)$.

Throughout the present part, we manipulate real numbers which are the evaluation of expressions. These expressions belong to a set $\mathcal{E}$ built from π, $n \in \mathbb{Z}$, a countable set of variables x, y, z, $\cdots$, the addition $(+)$, the product $(\cdot)$, division $(/)$, the inverse $(\cdot^{-1})$, the absolute value $(|\cdot|)$, the integer power $(\cdot^n)$, the exponential function (exp), the natural logarithm function (ln), and the trigonometric functions cos, sin, tan, and arctan.

Fix an enumeration of the variables. We define a function $eval_{\mathbb{R}}$ in the following way. Let $s \in \mathbb{N}$. The function $eval_{\mathbb{R}}$ takes as input an expression $e \in \mathcal{E}$, a vector $\mathbf{x} = (x_1, \cdots, x_s) \in \mathbb{R}^s$, and it returns the evaluation $eval_{\mathbb{R}}(e, \mathbf{x})$ of the expression by following inductively the structure of e, applying the corresponding real operation, and replacing the i-th variable with x_i. Whenever an operation is not defined or a variable of index greater than s appears, the function $eval_{\mathbb{R}}$ returns an error $\perp$.

When there is no ambiguity, we identify a variable-free expression with its evaluation. When we write "$e(x) \in \mathcal{E}$", this means that $e(x)$ is an expression whose only variable is x. If there is no ambiguity, we write $t \mapsto e(t)$ for the function $t \mapsto eval_{\mathbb{R}}(e, t)$, and if $t \in \mathbb{R}$, we write $e(t)$ for $eval_{\mathbb{R}}(e, t)$.

11.1.2 Interval Arithmetic

The paradigm of interval arithmetic

As the name indicates, interval arithmetic is a paradigm of computation in which, instead of manipulating real values, one handles intervals of real values. For example, we might represent the number π by the interval:

$$[3.14159; 3.14160].$$

One of the goals of interval arithmetic is, given a real number $x \in \mathbb{R}$ presented as an expression $e \in \mathcal{E}$, to build an interval $\mathbf{x}$ such that

$$x \in \mathbf{x}. \tag{11.1}$$

Naive interval arithmetic

There are several ways to implement interval arithmetic. In the simplest one called *naive* interval arithmetic, usual unary and binary operations are overloaded to the realm of intervals and an enclosure of an expression e is built by composing these overloaded operators following the inductive structure of e. The following *inclusion properties* ensure that equation (11.1) holds by an induction on the structure of $e \in \mathcal{E}$.

- Given a unary operator $\diamond$ on real numbers, naive interval arithmetic provides a unary operator $\Diamond$ on intervals such that

$$\forall x \in \mathbb{R},\ \forall \mathbf{x} \in \mathbb{I},\ x \in \mathbf{x} \Rightarrow \diamond(x) \in \Diamond(\mathbf{x}). \tag{11.2}$$

  $\Diamond$ is said to be an *interval extension* of $\diamond$.

An example would be the exponential function: if $\diamond = \exp$, define

$$\begin{aligned}\mathbf{Exp} &: \mathbb{I} \to \mathbb{I} \\ [a, b] &\mapsto [\exp(a), \exp(b)].\end{aligned}$$

Since exp is an increasing function, $\Diamond := \mathbf{Exp}$ satisfies (11.2).

- Given a binary operator $\diamond$ on real numbers, naive interval arithmetic provides a binary operator $\Diamond$ on intervals such that

$$\forall x, y \in \mathbb{R},\ \forall \mathbf{x}, \mathbf{y} \in \mathbb{I},\ x \in \mathbf{x} \wedge y \in \mathbf{y} \Rightarrow x \diamond y \in \mathbf{x} \Diamond \mathbf{y}. \tag{11.3}$$

  In this binary case, we also say that $\Diamond$ is an interval extension of $\diamond$.

  If we take $\diamond$ to be the multiplication operation, then $\cdot_{\mathbb{I}}$ defined in Section 11.1.1 satisfies (11.3).

Informative Extensions

An interval extension can be more or less informative. In the unary case of a function $f : \mathbb{R} \to \mathbb{R}$, the most informative extension F is the convex hull of the image of f:

$$\begin{aligned}F &: \mathbb{I} \to \mathbb{I} \\ \mathbf{x} &\mapsto \mathrm{hull}(\{f(t) \mid t \in \mathbf{x}\})\end{aligned}$$

Thanks to (11.2), for any interval extension G of f and any $\mathbf{x}$, $F(\mathbf{x}) \subseteq G(\mathbf{x})$.

In the other direction, the *least* informative interval extension of a total function $f : \mathbb{R} \to \mathbb{R}$ is certainly

$$\begin{aligned}F &: \mathbb{I} \to \mathbb{I} \\ \mathbf{x} &\mapsto (-\infty, +\infty).\end{aligned}$$

Correlation

The approach of naive interval arithmetic cannot keep track of correlations between subexpressions and might compute overestimated enclosures which are thus useless for proving some goals. This is true even in the case when each operator in the definition of $\mathcal{E}$ is overloaded with the most informative interval extension.

For instance, assume that $x \in [3; 4]$, so $-x \in [-4; -3]$ using the (optimal) interval extension of the negation, so $x + (-x) \in [3 + (-4); 4 + (-3)]$ using the (optimal) interval extension of the addition. If the goal was to prove that $x - x$ is always 0, the interval $[-1; 1]$ obtained by naive interval arithmetic is useless.

11.1.3 Interval Arithmetic and CoqInterval

The CoqInterval library

CoqInterval is a Coq library for computing numerical enclosures of real-valued expressions [91]. It also provides a tactic `interval` to automatically deduce certain goals from these enclosures.

The tactic typically takes a goal $A \leq e \leq B$ where e is an expression in $\mathcal{E}$, and A and B are constants. Using the paradigm of interval arithmetic described in Section 11.1.2, it builds a set $\mathbf{e}$ such that $e \in \mathbf{e}$ holds by construction and such that $\mathbf{e}$ evaluates to an interval $[\inf \mathbf{e}; \sup \mathbf{e}]$ by computation. Then it checks that $A \leq \inf \mathbf{e}$ and $\sup \mathbf{e} \leq B$, again by computation, from which it proves $A \leq e \leq B$. All the computations on interval bounds are performed using a rigorous yet efficient formalization of multi-precision floating-point arithmetic.

Because of the problem of correlation described in Section 11.1.2, the CoqInterval library also comes with refinements of naive interval arithmetic, such as automatic differentiation and rigorous polynomial approximations, so as to reduce this loss of correlations.

Data structures for reals and intervals

CoqInterval defines $\overline{\mathbb{R}}$ as the set $\mathbb{R} \cup \{\bot_{\mathbb{R}}\}$ of *extended reals*, that is the set of real numbers completed with the extra point $\bot_{\mathbb{R}}$. This is an option type: an element of $\overline{\mathbb{R}}$ is either a real number r, or it is a “default value” $\bot_{\mathbb{R}}$. The value $\bot_{\mathbb{R}}$ is typically outputted when an operation is applied outside of its usual domain of definition. For instance, the result of dividing one by zero in $\overline{\mathbb{R}}$ is $\bot_{\mathbb{R}}$, while it is unspecified in $\mathbb{R}$. This $\bot_{\mathbb{R}}$ element is then propagated along the subsequent operations.

Similarly, CoqInterval defines a type $\overline{\mathbb{I}} = \mathbb{I} \cup \{\bot_{\mathbb{I}}\}$ of extended intervals. 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. In other words, an interval operator produces $\bot_{\mathbb{I}}$ whenever the corresponding operator on $\overline{\mathbb{R}}$ would have produced $\bot_{\mathbb{R}}$ for at least one value in one of the input intervals. Just like $\bot_R$, $\bot_I$ propagates along computations. The $\cdot \in \cdot$ membership relation is hence extended:

Definition 12 (membership in $\overline{\mathbb{R}}$ and $\mathbb{I}$)**.** *The elements $\bot_{\mathbb{R}}$ and $\bot_{\mathbb{I}}$ have the following properties with regard to membership:*

- $\bot_{\mathbb{R}} \notin [u, v]$ *for all* u, $v \in \mathbb{R} \cup \{-\infty, +\infty\}$;
- $\bar{x} \in \bot_{\mathbb{I}}$ *for all* $\bar{x} \in \overline{\mathbb{R}}$.

In particular, a crucial remark is that $\bot_{\mathbb{I}}$ is the only interval containing $\bot_{\mathbb{R}}$.

The implementation of interval arithmetic on which we build uses pairs of floating-point numbers $[\inf \mathbf{x}; \sup \mathbf{x}]$. Unless otherwise specified, we will be identifying $\mathbb{I}$ and $\overline{\mathbb{I}}$ from now on.

11.1.4 Representation of functions, efficient computations: straight-line programs

The CoqInterval library represents expressions in $\mathcal{E}$ as straight-line programs. This allows for explicit sharing of common subexpressions. Such a program is just a list of statements indicating what the operation is and where its inputs can be found. The place where the output is stored is left implicit: the result of an operation is always put at the top of the evaluation stack. The evaluation model is simple: the stack grows linearly with the size of the expression since no element of the stack is ever removed. The stack is initially filled with values

corresponding to the constants of the program. The result of evaluating a straight-line program is at the top of the stack.

Below is an example of a straight-line program corresponding to the expression $(t + \pi)\sqrt{t} - (t + \pi)$. It is a list containing the operations to be performed. Each list item first indicates the arity of the operation, then the operation itself, and finally the relative depth at which the inputs of the operation can be found in the evaluation stack. Note that, in this example, t and π are seen as constants, so the initial stack contains values that correspond to these subterms[1]. The comments in the term below indicate the content of the evaluation stack before evaluating each statement.

```
(* initial stack: [t, pi]                    *)     Binary Add 0 1
(* current stack: [t+pi, t, pi]              *)  :: Unary Sqrt 1
(* current stack: [sqrt t, t+pi, t, pi]      *)  :: Binary Mul 1 0
(* current stack: [(t+pi)*sqrt t, sqrt t, ...]  *)  :: Binary Sub 0 2
(* current stack: [(t+pi)*sqrt t - (t+pi), ...] *)  :: nil
```

The evaluation of a straight-line program depends on the interpretation of the arithmetic operations and on the values stored in the initial stack. For instance, if the arithmetic operations are the operations from the `Reals` library (*e.g.* `Rplus`) and if the stack contains the symbolic value of the constants, then the result is the actual expression over real numbers. If u is a unary operator, we will denote by $[\![u]\!]_{\mathbb{R}}$ the real interpretation of u. So if u is the `Sqrt` operator, $[\![u]\!]_{\mathbb{R}} = \sqrt{\cdot}$. Similarly, if b is a binary operator, then $[\![b]\!]_{\mathbb{R}}$ will denote its real interpretation, so that if b is `Add`, $[\![b]\!]_{\mathbb{R}} = +$. We will use a similar notation $[\![u]\!]_A$ and $[\![b]\!]_A$ for other evaluation schemes into a set A.

Let us denote $[\![p]\!]_{\mathbb{R}}(\vec{x})$ the result of evaluating the straight-line program p with operators from `Reals` over an initial stack $\vec{x}$ of real numbers. Similarly, $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ denotes the result of evaluating p with interval operations over a stack of intervals. Then, thanks to the inclusion property of interval arithmetic, we can prove the following formula once and for all:

$$\forall p,\ \forall \vec{x} \in \mathbb{R}^n,\ \forall \vec{\mathbf{x}} \in \mathbb{I}^n,\ (\forall i \le n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\mathbb{R}}(\vec{x}) \in [\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}). \tag{11.4}$$

In particular, Formula (11.4) implies that if a function f can be reified into $t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x})$, then $\mathbf{t} \mapsto [\![p]\!]_{\mathbb{I}}(\mathbf{t}, \vec{\mathbf{x}})$ is an interval extension of f if $\forall i,\ x_i \in \mathbf{x}_i$.

Formula (11.4) is the basic block used by the `interval` tactic for proving enclosures of expressions [91]. Given a goal $A \le e \le B$, the tactic first looks for a program p and a stack $\vec{x}$ of real numbers such that $[\![p]\!]_{\mathbb{R}}(\vec{x}) = e$. This process of finding a program p is called *reification*. It is done by a tactic outside the logic of Coq, so that the fact that $[\![p]\!]_{\mathbb{R}}(\vec{x}) = e$ holds is checked by Coq. More precisely, the goal $A \le e \le B$ is convertible to $[\![p]\!]_{\mathbb{R}}(\vec{x}) \in [A; B]$ if A and B are floating-point numbers and if the tactic successfully reified the term.

The tactic then looks in the context for hypotheses of the form $A_i \le x_i \le B_i$ so that it can build a stack $\vec{\mathbf{x}}$ 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 $\mathbf{x}_i$. The tactic can now apply Formula (11.4) to replace the goal by $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}) \subseteq [A; B]$. 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 as seen in Section 11.1.2.

[1] The only thing that will later distinguish the integration variable t from an actual constant such as π is that its value is placed at the top of the initial evaluation stack.

11.2 Rigorous Polynomial Approximations

11.2.1 CoqApprox: Rigorous Polynomial Approximations and Taylor Models in Coq

Although naive interval arithmetic is useful for approximating real expressions, we have seen in Section 11.1.2 that it suffers from loss of correlation. A less crude way of approximating real numbers on an interval is the use of uniform approximations of functions by polynomials:

Definition 13 (Rigorous Polynomial Approximations). *Let $\mathbf{x} \in \mathbb{I}$ and let $f : \mathbf{x} \to \mathbb{R}$ be a function which is n+1 times differentiable.*

A rigorous polynomial approximation (RPA) for a f is a pair $(P, \mathbf{\Delta})$ where $P \in \mathbb{R}_n[X]$ and $\mathbf{\Delta} \in \mathbb{I}$, such that

$$\forall x \in \mathbf{x}, f(x) - P(x) \in \mathbf{\Delta}.$$

RPAs as defined in Definition 13 are agnostic as to the way in which they are obtained. However, the usual privileged method of producing them is by truncating a Taylor series expansion of the function f around a point $x_0 \in \mathbf{x}$. This method uses an interval version of the Taylor-Lagrange formula to estimate the remainder:

$$\forall x \in \mathbf{x}, f(x) - \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i \in \frac{f^{(n+1)}(\mathbf{x} - x_0)}{(n+1)!}(\mathbf{x} - x_0)^{n+1} \qquad (11.5)$$

Such a RPA is called a Taylor Model [90]. The CoqApprox [91] library is an extension to CoqInterval which provides a generic theory of RPAs. It also comes with an formalized effective implementation which can build Taylor-Model-based RPAs for univariate expressions in the set $\mathcal{E}$ described in Section 11.1.1. The formalization is based on the algorithms developed in the PhD thesis of Joldes [77].

Since this formalization contains an effective version of RPAs using interval arithmetic, the actual data types used during computations are different from those in the idealized context of Definition 13: polynomials have interval coefficients, f takes its values in $\overline{\mathbb{R}}$ to accommodate for its domain, x_0 becomes an interval $\mathbf{X_0}$. The CoqApprox library defines a predicate specifying what it means for such a Rigorous Polynomial Approximation to be correct with respect to a function it approximates:

Definition 14 (Validity predicate for RPAs in CoqApprox). *Let $\mathbf{P} := \sum_{i=0}^{n} \alpha_i X^i$ be a degree-n polynomial with interval coefficients and let $\mathbf{\Delta} \in \mathbb{I}$. Then $(P, \mathbf{\Delta})$ is a valid RPA for $f : \mathbb{R} \to \overline{\mathbb{R}}$ on $\mathbf{I}$ around $\mathbf{X_0}$ if*

1. *f is defined on $\mathbf{I}$, that is,*

$$\forall x \in \mathbf{I}, (f(x) = \bot_{\mathbb{R}} \implies \mathbf{\Delta} = \bot_{\mathbb{I}});$$

2. *$\mathbf{I} = \bot_{\mathbb{I}} \implies \mathbf{\Delta} = \bot_{\mathbb{I}}$;*
3. *$0 \in \mathbf{\Delta}$;*

4. $\mathbf{X_0} \subseteq \mathbf{I}$*;*

5. $\forall x_0 \in \mathbf{X_0}, \exists Q \in \mathbb{R}_n[X],$
$Q \in \mathbf{P} \wedge \forall x \in \mathbf{I}, f(x) - Q(x - x_0) \in \mathbf{\Delta}.$

where $Q \in \mathbf{P}$ *means that if* $Q = \sum_{i=0}^{n} a_k X^k$,

$$\forall i, 0 \leq i \leq n \implies a_i \in \alpha_i.$$

11.2.2 Primitives of RPAs

The following work is a contribution and was integrated into CoqInterval.

We place ourselves within the context of Definition 13 in order not to make the exposition cumbersome with interval coefficient polynomials. If $T = (P, \mathbf{I})$ is a valid RPA for $f : \mathbf{X} \to \mathbb{R}$ with respect to $\mathbf{X_0} \subset \mathbf{X}$, we want to build a valid RPA $U = (Q, \mathbf{J})$ for $x \mapsto \int_{x_0}^{x} f(s)ds$ with respect to the same $\mathbf{X_0}$,$\mathbf{X}$. We write $P = \sum_{k=0}^{n} a_k X^k$, and for all $c \in \mathbb{R}$, we define $\text{prim}(P, c)$ by

$$\text{prim}(P, c) = c + \sum_{k=0}^{n} \frac{a_k}{k+1} X^{k+1}$$

In other words, $\text{prim}(P, c)$ is the only primitive of P which evaluates to c at 0. Consider $R = \text{prim}(P, 0)$. If $x_0, x_1 \in \mathbf{X_0}$, then

$$\begin{aligned}
\forall x \in \mathbf{X}, \int_{x_0}^{x} f(s)\,ds - R(x - x_1) &= \int_{x_0}^{x} f(s)\,ds - [R(s - x_1)]_{x_1}^{x} \\
&= \int_{x_0}^{x} f(s)\,ds - \int_{x_1}^{x} P(s - x_1)\,ds \\
&= \int_{x_1}^{x} (f(s) - P(s - x_1))\,ds + \int_{x_0}^{x_1} f(s)\,ds
\end{aligned}$$

but for $s \in [x_0, x_1]$[2], thanks to Definition 13,

$$f(s) - P(s - x_0) \in I$$

so that

$$f(s) \in P(s - x_0) + I$$

and then

$$\int_{x_0}^{x_1} f(s) \in \int_{x_0}^{x_1} P(s - x_0)\,ds + (\mathbf{X_0} - \mathbf{X_0})\mathbf{I}$$

and thus

$$\int_{x_0}^{x_1} f(s)\,ds \in R(\mathbf{X_0} - \mathbf{X_0}) + (\mathbf{X_0} - \mathbf{X_0})\mathbf{I}$$

which brings us to the inclusion:

$$\forall x \in X, \int_{x_0}^{x} f(s)\,ds - R(x - x_1) \in (x - x_0)\mathbf{I} + R(\mathbf{X_0} - \mathbf{X_0}) + (\mathbf{X_0} - \mathbf{X_0})\mathbf{I}$$

[2] where $[x_0, x_1]$ represents the interval whose ends are x_0 and x_1 irrespective of their relative order. This interval is included in X_0.

and finally in all generality

$$\forall x \in \mathbf{X}, \int_{x_0}^{x} f(s)\, ds - R(x - x_1) \in (\mathbf{X} - \mathbf{X_0})\mathbf{I} + R(\mathbf{X_0} - \mathbf{X_0}) + (\mathbf{X_0} - \mathbf{X_0})\mathbf{I}$$

so that $(R, (\mathbf{X}-\mathbf{X_0})I+R(\mathbf{X_0}-\mathbf{X_0})+(\mathbf{X_0}-\mathbf{X_0})\mathbf{I})$ is an RPA for $x \mapsto \int_{x_0}^{x} f(s)\, ds$ on $(\mathbf{X_0}, \mathbf{X})$.

Notice that if $\mathbf{X_0}$ is thin (which is a reasonable scenario, as we are developing around $\mathbf{X_0}$), the error reduces to $(\mathbf{X} - \mathbf{X_0})\mathbf{I}$ which is simply the scaling of the error by the size of the interval.

Chapter 12

Computing Estimates of Integrals in Coq

12.1 Estimating Proper Integrals

In this section, we describe how to compute a numerical enclosure of the real number $\int_u^v f$ from enclosures of the finite bounds u and v and of the Riemann-integrable function f. In other words, we only deal with *proper* integrals: the case of improper integrals will be handled later in Section 12.2. We describe two basic methods based respectively on the evaluation of a simple interval extension and on a polynomial approximation of f. They can be combined and improved by a dichotomy process.

This is integrated into the `interval` tactic of CoqInterval.

12.1.1 A Naive Approach

Our first approach uses an *interval extension* of the integrand. Let us formalize and extend this notion already mentioned in Section 11.1.2.

Definition 15. *For any function* $f : \mathbb{R}^n \to \mathbb{R}$*, a function* $\mathbf{F} : \mathbb{I}^n \to \mathbb{I}$ *is an* interval extension *of* f *on* $\mathbb{R}$ *if*

$$\forall \mathbf{x}_1, \ldots, \mathbf{x}_n,\ \{f(x_1, \ldots, x_n) \mid \forall i, x_i \in \mathbf{x}_i\} \subseteq \mathbf{F}(\mathbf{x}_1, \ldots, \mathbf{x}_n).$$

In the rest of the section we suppose that $\mathbf{F} : \mathbb{I} \to \mathbb{I}$ is an interval extension of the univariate function f, and we want to compute an enclosure of $\int_u^v f$, with $u, v \in \mathbb{R}$, and f integrable on $[u; v]$.

Lemma 25 (Naive integral enclosure)**.**

$$\int_u^v f \in (v - u) \cdot \mathrm{hull}\{f(t) \mid t \in [u; v] \text{ or } t \in [v; u]\}. \tag{12.1}$$

Proof. Let us first suppose that $u \leq v$. Denote $f([u; v]) := \{f(t) \mid t \in [u; v]\}$. Assume without loss of generality that $f([u; v])$ is bounded. If $[m; M] := \mathrm{hull}(f([u; v]))$, then for any $t \in [u; v]$, we have $m \leq f(t) \leq M$. So $(v - u)m \leq \int_u^v f \leq (v - u)M$, hence Formula (12.1). The case $v \leq u$ is symmetrical. □

In practice we do not compute with f but only with its interval extension $\mathbf{F}$. Moreover, we want the computations to operate using only enclosures of the bounds. So we adapt Formula (12.1) accordingly.

Lemma 26 (Interval naive integral enclosure)**.** *For any intervals* $\mathbf{u}, \mathbf{v}$ *such that* $u \in \mathbf{u}$ *and* $v \in \mathbf{v}$*, we have*

$$\int_u^v f \in (\mathbf{v} - \mathbf{u}) \cdot \mathbf{F}(\mathrm{hull}(\mathbf{u}, \mathbf{v})). \tag{12.2}$$

Note that if $\mathbf{u}$ *and* $\mathbf{v}$ *are point intervals and if* $\mathbf{F}$ *is the optimal interval extension of* f*, then equation* (12.2) *reduces to equation* (12.1)*.*

Proof. If $u \in \mathbf{u}$ and $v \in \mathbf{v}$, then by (12.1) and reusing notations from the proof, we have $\int_u^v f \in (v - u) \cdot \mathrm{hull}(f([u; v]))$. Since $(v - u) \in (\mathbf{v} - \mathbf{u})$, we only have to show that $\mathrm{hull}(f([u; v])) \subseteq F(\mathrm{hull}(\mathbf{u}, \mathbf{v}))$. Since $[u; v] \subseteq \mathrm{hull}(\mathbf{u}, \mathbf{v})$ and F is an interval extension of f, we have $f([u; v]) \subseteq f(\mathrm{hull}(\mathbf{u}, \mathbf{v})) \subseteq F(\mathrm{hull}(\mathbf{u}, \mathbf{v}))$. Therefore $\mathrm{hull}(f([u; v]))$ is included in the interval $F(\mathrm{hull}(\mathbf{u}, \mathbf{v}))$, by definition of the closed convex hull. □

The `naive_integral` Coq function implements (12.2). Given $\mathbf{u}, \mathbf{v} \in \mathbb{I}$ and F a function of type $\mathbb{I} \to \mathbb{I}$, `(naive_integral prec F u v)` computes an interval $\mathbf{i}$ using floating-point arithmetic at precision `prec`. If F is an interval extension of f, if $u \in \mathbf{u}$ and $v \in \mathbf{v}$, and if f is integrable on $[u; v]$, then $\int_u^v f \in \mathbf{i}$.

12.1.2 Another Approach using RPAs

The enclosure method described in Section 12.1.1 is crude. Better knowledge of the integrated function allows for a more efficient approach. The goal of this section is to use RPAs introduced in Section 11.2.1 to obtain better enclosures of integrals.

Lemma 27 (Polynomial approximation)**.** *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. We have $\int_u^v f - (P(v) - P(u)) = \int_u^v (f(t) - p(t))\,dt$. By hypothesis, the constant function $\mathbf{\Delta}$ is an interval extension of $t \mapsto f(t) - p(t)$ on $[u; v]$, hence Lemma 25 applies (notice that $\mathrm{hull}(\mathbf{\Delta}) = \mathbf{\Delta}$). □

Note that our method and proofs do not depend on the way polynomial approximations are obtained.

Incidentally, we recently realized while reading Rall and Corliss [34] that by putting $c = \frac{u+v}{2}$ and choosing p to be of the shape $\sum_{i=0}^{n} a_i (x - c)^i$, P can be chosen so that $P(v) - P(u)$ can be computed in half as many operations. We plan to take benefit from this remark in our code in the near future.

12.1.3 Polynomial Approximations Are Not Always Better

Polynomial approximations usually give tighter enclosures of an integral $\int_u^v f$ than the naive approach. In fact, the naive approach is an instance of the polynomial one: let y be the midpoint of $\mathbf{F}(\text{hull}(\mathbf{u}, \mathbf{v}))$ and let $\mathbf{\Delta} := \mathbf{F}(\text{hull}(\mathbf{u}, \mathbf{v})) - y$. Then $(y, \mathbf{\Delta})$ is a polynomial approximation of degree 0 for f since for all $t \in [u; v]$, $f(t) \in \mathbf{F}(\text{hull}(\mathbf{u}, \mathbf{v}))$ and thus $f(t) - y \in \mathbf{\Delta}$.

However, due to the way polynomial approximations are obtained in CoqApprox, it can happen that a polynomial approximation is less accurate than the use of naive interval arithmetic.

One first instance could be around a singularity preventing the Taylor series of f to exist at all points of $[u; v]$, as in the example of $x \mapsto \sqrt{x}$ on $[0; \varepsilon]$, with $\varepsilon > 0$. In this case, CoqApprox returns a pair $(P, \bot_{\mathbb{I}})$ which yields the uninformative enclosure $\bot_{\mathbb{I}}$ for $\int_0^\varepsilon \sqrt{x}\,dx$, whereas the naive estimate would have given $(\varepsilon - 0) \cdot [0; \sqrt{\varepsilon}] = [0; \varepsilon\sqrt{\varepsilon}]$.

Another instance would be if we wanted to integrate a strongly oscillating function over a large interval. In Section 12.3, we will be looking at the integral

$$\int_0^8 \sin(x + \exp(x))\,dx$$

Indeed, let us use the software Sollya [24], which implements the same algorithm as CoqApprox, to find a Taylor Model for the integrand on the whole interval $[0; 8]$. In Figure 12.1, we use the command `taylorform` which takes as input the function f, the approximating degree n, the point around which to develop the Taylor series x_0, and the interval $\mathbf{I}$ on which to find the approximation. It returns a Taylor model $(P, \mathbf{\Delta})$. We only look at $\mathbf{\Delta}$ for our purposes here, in the cases of $n = 10$ and $n = 50$.

Figure 12.1: Sollya Session

```
> TL=taylorform(sin(x + exp(x)), 10, 0, [0,8]);
> TL[2];
[-4.2589929933332345326623361369068740662104390570317e30;
  4.2458685067031149053103989041910036815095504573458e30]

> TL=taylorform(sin(x + exp(x)), 50, 0, [0,8]);
> TL[2];
[-1.14722088059037692970526490071119314283016871650084e111;
  1.13115598945164611003185544497692438747503367195052e111]
```

In the case of $n = 10$, the error $\mathbf{\Delta}$ has width greater than $2 \cdot 10^{30}$; when we try to increase the degree to 50, it is greater than $2 \cdot 10^{111}$. The reason is that the n-th derivative of this function is big and the error interval obtained from Equation (11.5) is not tight. Lemma 27 is going to yield a useless estimate. Now if we use a naive estimate, we get the much better enclosure

$$\int_0^8 \sin(x + \exp(x))\, dx \in 8 \cdot [-1; 1] = [-8; 8].$$

12.1.4 Dichotomy

Both methods presented in Sections 12.1.1 and 12.1.2 can compute an interval enclosing $\int_u^v f$ when u and v are proper bounds. As seen in Section 12.1.3, polynomial approximations usually give tighter enclosures of the integral, but not always, so we combine both methods by taking the intersection of their result.

This may still not be sufficient for getting a tight enough enclosure, in which case we recursively split the integration domain in two parts, adding up the results. The function `integral_float_absolute` performs this dichotomy and the integration on each subdomain. It takes an absolute error parameter ε; it stops splitting as soon as the width of the computed integral enclosure is smaller than ε. The function also takes a *depth* parameter, which means that the initial domain is split into at most $2^{depth+1}$ subdomains. Note that, because the depth is bounded, there is no guarantee that the target width will be reached.

Let us detail more precisely how the function behaves. It starts by splitting $[u; v]$ into $[u; m]$ and $[m; v]$ where $m = \frac{u+v}{2}$. It then computes some enclosures $\mathbf{i_1}$ of $\int_u^m f$ and $\mathbf{i_2}$ of $\int_m^v f$. If $depth = 0$, the function returns $\mathbf{i_1} + \mathbf{i_2}$. Otherwise, several cases can occur:

- If $w(\mathbf{i_1}) \le \frac{\varepsilon}{2}$ and $w(\mathbf{i_2}) \le \frac{\varepsilon}{2}$, the function simply returns $\mathbf{i_1} + \mathbf{i_2}$.
- If $w(\mathbf{i_1}) \le \frac{\varepsilon}{2}$ and $w(\mathbf{i_2}) > \frac{\varepsilon}{2}$, the first enclosure is sufficient but the second is not. So `integral_float_absolute` calls itself recursively on $[m; v]$ with $depth - 1$ as the new maximal depth and $\varepsilon - w(\mathbf{i_1})$ as the new target accuracy, yielding $\mathbf{i_2'}$. The function then returns $\mathbf{i_1} + \mathbf{i_2'}$.
- If $w(\mathbf{i_1}) > \frac{\varepsilon}{2}$ and $w(\mathbf{i_2}) \le \frac{\varepsilon}{2}$, we proceed symmetrically.
- Otherwise, the function calls itself on $[u; m]$ and $[m; v]$ with $depth - 1$ as the new maximal depth and $\frac{\varepsilon}{2}$ as the new target accuracy, yielding $\mathbf{i_1'}$ and $\mathbf{i_2'}$. It then returns $\mathbf{i_1'} + \mathbf{i_2'}$.

12.1.5 Accuracy

Both methods described in Sections 12.1.1 and 12.1.2 use a single approximation of the integrand on the integration interval. A decomposition of this interval into smaller pieces may increase the accuracy of the enclosure, if tighter approximations are obtained on each subinterval. In this section we give an intuition of how the naive and polynomial approaches compare, from a time complexity point of view. What we mean here by complexity is the number of operations needed to obtain a result of width less than a parameter $0 < \varepsilon$ given as an input to the algorithm.

The naive (resp. polynomial) approach here consists in using a simple interval approximation (resp. a valid polynomial approximation) to estimate the integral on each subinterval. Let us suppose that we split the initial integration interval, before computing integral enclosures:

$$\int_u^v f = \int_{x_0}^{x_1} f + \ldots + \int_{x_{n-1}}^{x_n} f \quad \text{with } x_i = u + \tfrac{i}{n}(v-u).$$

Let $w(\mathbf{x}) = \sup \mathbf{x} - \inf \mathbf{x}$ denote the width of an interval. The smaller $w(\mathbf{x})$ is, the more accurately any real $x \in \mathbf{x}$ is approximated by $\mathbf{x}$. Any sensible interval arithmetic respects $w(\mathbf{x}+\mathbf{y}) \simeq w(\mathbf{x}) + w(\mathbf{y})$ and $w(k \cdot \mathbf{x}) \simeq k \cdot w(\mathbf{x})$.

We consider the case of the naive approach first. We assume that F is an optimal interval extension of f and that f has a Lipschitz-constant equal to k_0, that is, $w(F(\mathbf{x})) \simeq k_0 \cdot w(\mathbf{x})$. Since $w(\text{naive}([x_i; x_{i+1}])) \simeq (x_{i+1} - x_i) \cdot w(F([x_i; x_{i+1}]))$, we get the following accuracy when computing the integral:

$$w\left(\sum_i \text{naive}([x_i; x_{i+1}])\right) \simeq \sum_i (x_{i+1} - x_i) \cdot k_0 \cdot w(([x_i; x_{i+1}])) \tag{12.3}$$

$$\simeq k_0 \cdot (v-u)/n \cdot \sum_i w(([x_i; x_{i+1}])) \tag{12.4}$$

$$\simeq k_0 \cdot (v-u)^2/n. \tag{12.5}$$

Suppose we want to gain one bit of accuracy by going from n_1 integrals in the above sum to some number n_2 of integrals. This means that we want $k_0 \cdot (v-u)^2/n_2 \leq \frac{1}{2} \cdot k_0 \cdot (v-u)^2/n_1$, so that $n_2 \geq 2\, n_1$, so that we have to multiply the computation time by at least two, hence the complexity is exponential.

Now for the polynomial enclosure. Let us assume we can compute a polynomial approximation of f on any interval $\mathbf{x}$ with an error $\mathbf{\Delta}(\mathbf{x})$. Because of Formula (11.5), we can expect this error to satisfy $w(\mathbf{\Delta}(\mathbf{x})) \simeq k_d \cdot w(\mathbf{x})^{d+1}$ with d the degree of the polynomial approximation and k_d depending on f. Since

$$w(\text{poly}([x_i; x_{i+1}])) \simeq (x_{i+1} - x_i) \cdot w(\mathbf{\Delta}([x_i, x_{i+1}])),$$

developing as in (12.3), the accuracy is now

$$w\left(\sum_i \text{poly}([x_i; x_{i+1}])\right) \simeq k_d \cdot (v-u)^{d+2}/n^{d+1}.$$

For a fixed d, $k_d \cdot (v-u)^{d+2}/n_2^{d+1} \leq \frac{1}{2} \cdot k_d \cdot (v-u)^{d+2}/n_1^{d+1}$ implies that $n_2 \geq 2^{\frac{1}{d+1}} n_1$: one still has to increase n exponentially with respect to the target accuracy. The power coefficient, however, is much smaller than for the naive method. By doubling the computation time, one gets $d+1$ additional bits of accuracy.

In order to improve the accuracy of the result, one can increase d instead of n. If f behaves similarly to exp or sin, Taylor-Lagrange formula tells us that k_d decreases as fast as $(d!)^{-1}$. Moreover, the time complexity of computing a polynomial approximation usually grows like d^3. If e.g. $n \simeq v - u$, doubling the computation time is thus done by increasing d to d' by a factor of about $2^{\frac{1}{3}} \simeq 1.25$. This gives about 25% more bits of accuracy.

As can be seen from the considerations above, striking the proper balance between n and d for reaching a target accuracy in a minimal amount of time is difficult, so we have made the decision of letting the user control d (see Section 12.1.4) while the implementation adaptively splits the integration interval.

12.1.6 Automatic Continuity and Integrability

When computing the enclosure of an integral, the tactic should first obtain a formal proof that the integrand is integrable on the integration domain, as this is a prerequisite to all the theorems in Section 12. In fact we can be more clever: we prove that, if we succeed in numerically computing an *informative* enclosure of the integral, the function was actually integrable. This way, the tactic does not have to prove anything beforehand about the integrand.

For this, we use the data type for extended reals $\overline{\mathbb{R}} = \mathbb{R} \cup \{\bot_{\mathbb{R}}\}$ defined in Section 11.1.3. In order to gain intuition of what we are trying to do, let us look at a simple example first.

Suppose we have a function $\overline{\ln} : \overline{\mathbb{R}} \to \overline{\mathbb{R}}$ such that

$$\forall x \in \mathbb{R}, x > 0 \implies \overline{\ln}\, x = \ln x; \tag{12.6}$$

$$\forall x \in \mathbb{R}, x \leq 0 \implies \overline{\ln}\, x = \bot_{\mathbb{R}}; \tag{12.7}$$

$$\overline{\ln}(\bot_{\mathbb{R}}) = \bot_{\mathbb{R}}. \tag{12.8}$$

Suppose that $\mathbf{ln} : \mathbb{I} \to \mathbb{I}$ is an extension of $\overline{\ln}$ and $\mathbf{ln}([u; v]) \neq \bot_{\mathbb{I}}$, that is $\mathbf{ln}([u; v]) = [u'; v']$ for some u', $v' \in \mathbb{R}$. Since $\bot_{\mathbb{I}} \not\subseteq [u'; v']$, $\bot_{\mathbb{R}} \notin [u'; v']$ because of Definition 12. Furthermore, we can deduce from Equations (12.6), (12.7) and (12.8) that $[u; v] \subseteq (0; +\infty)$. Hence ln is defined on $[u; v]$, and since ln is continuous on its domain of definition, we can deduce that ln is continuous on $[u; v]$. We just deduced *from computation* that a function was continuous. As any continuous function on a compact interval is also integrable on that interval, we will be using this principle to automatically deduce integrability.

Let us make this more formal. We already saw two evaluation schemes $[\![p]\!]_{\mathbb{R}}$ and $[\![p]\!]_{\mathbb{I}}$ in Section 11.1.4. The alternate scheme $[\![p]\!]_{\overline{\mathbb{R}}}$ produces the value $\bot_{\mathbb{R}}$ as soon as an operation is applied to inputs that are outside the usual definition domain of the operator (this would be a way to define $\overline{\ln}$ in the example above). This $\bot_{\mathbb{R}}$ element is then propagated along the subsequent operations. Thus, the following equality holds, using the trivial embedding from $\mathbb{R}$ into $\overline{\mathbb{R}}$:

$$\forall p,\ \forall \vec{x} \in \mathbb{R}^n,\ [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}) \neq \bot_{\mathbb{R}} \Rightarrow [\![p]\!]_{\mathbb{R}}(\vec{x}) = [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}). \tag{12.9}$$

Taking advantage of the properties of $\bot_{\mathbb{R}}$ and $\bot_{\mathbb{I}}$ in Definition 12, we can also prove a variant of Formula (11.4):

$$\forall p,\ \forall \vec{x} \in \overline{\mathbb{R}}^n,\ \forall \vec{\mathbf{x}} \in \mathbb{I}^n,\ (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}) \in [\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}). \tag{12.10}$$

In CoqInterval, Formula (11.4) is actually just a consequence of both Formulas (12.9) and (12.10). This is due to two properties of $\bot_{\mathbb{I}}$. First, $(-\infty; +\infty) \subseteq \bot_{\mathbb{I}}$ holds, so the conclusion of Formula (12.10) trivially holds whenever $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ evaluates to $\bot_{\mathbb{I}}$. Second, $\bot_{\mathbb{I}}$ is the only interval containing $\bot_{\mathbb{R}}$. As a consequence, whenever $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ does not evaluate to $\bot_{\mathbb{I}}$ the premise of Formula (12.9) holds.

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

domain. More precisely, we can prove the following property:

$$\forall p,\ \forall t_0 \in \mathbb{R},\ \forall \vec{x} \in \mathbb{R}^n,\ [\![p]\!]_{\overline{\mathbb{R}}}(t_0, \vec{x}) \neq \bot_{\mathbb{R}} \Rightarrow t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x}) \text{ is continuous at point } t_0. \quad (12.11)$$

By combining Formulas (11.4) and (12.11), we obtain a numerical method to prove that a function is continuous on a domain. Indeed, we just have to compute an enclosure of the function on that domain, and to check that it is not $\bot_{\mathbb{I}}$. A closer look at the way naive integral enclosures are computed provides the following corollary: whenever the enclosure of the integral is not $\bot_{\mathbb{I}}$, the function is actually continuous and thus integrable on any compact set of the input domain.

Note that Property (12.11) intrinsically depends on the operations that can appear inside p, *i.e.* the operations belonging to the class $\mathcal{E}$ of Section 11.1.2. Therefore, its proof has to be extended as soon as a new operator is supported in $\mathcal{E}$. This proof relies on the invariant:

Lemma 28 (Invariant on operators of $\mathcal{E}$)**.** *For all unary operators u included in the definition of $\mathcal{E}$, for all $x \in \mathbb{R}$:*

$$[\![u]\!]_{\overline{\mathbb{R}}}(x) \neq \bot_{\mathbb{R}} \implies [\![u]\!]_{\mathbb{R}} \textit{ is continuous at } x.$$

For all binary operators b included in the definition of $\mathcal{E}$, for all $x, y \in \mathbb{R}$:

$$[\![b]\!]_{\overline{\mathbb{R}}}(x, y) \neq \bot_{\mathbb{R}} \implies [\![b]\!]_{\mathbb{R}} \textit{ is continuous at } (x, y).$$

Suppose that we want to extend $\mathcal{E}$ with the integer part $\lfloor\cdot\rfloor$. Now Lemma 28 is not true any more. An unsolved question for now is: what more general invariant could be used to obtain integrability?

12.2 Estimating Improper Integrals

Following usual mathematical terminology, we qualify integrals as *improper* when their range is unbounded, as for $\int_1^{\infty} \frac{1}{x^2}\,dx$, or when the integrand is unbounded, such as $\int_0^1 \ln x\,dx$. They are defined as limits of proper integrals.

In this section, we describe how to compute a numerical enclosure of some improper integrals. Improper integrals are computed by splitting the interval into two parts, a proper part which is treated with the methods described previously in Section 12.1, and the remainder which is handled in a specific way. The splitting is automatically performed by a variant of the adaptive method presented in Section 12.1.4 where the splitting point m for $[u; +\infty)$ when $u > 0$ is chosen to be $2 \cdot u$. We now describe how we handle the remainder. On paper, when we want to show that an improper integral $\int_u^v f$ is well-defined, we typically show that

$$\forall t \in [u; v], |f(t)| \leq C \cdot g(t) \quad (12.12)$$

where $C > 0$ and g is a well-known integrable function such as $\frac{1}{x^2}$, $\ln x$ or $\exp(-x^2)$. On top of establishing that the integral exists, this leads to a natural way of bounding the integral:

$$\left|\int_u^v f(t)\,dt\right| \leq \int_u^v |f(t)|\,dt \leq C\int_u^v g(t)\,dt$$

so that

$$\int_u^v f(t)\,dt \in [-C;C]\int_u^v g(t)\,dt.$$

Establishing the bound (12.12) is not so easy to automate as it can involve complicated algebraic manipulations of usual functions. For this reason, we restrict ourselves to the case when this work has already been done for us. 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 $x^\alpha \ln^\beta x$.

In Sections 12.2.1 and 12.2.2 we focus on integrals of the shape $\int_u^{+\infty} fg$. Section 12.2.1 presents the general theorem, while Section 12.2.2 lists the functions contained in our catalog. Finally, Section 12.2.3 focuses on integrals of the shape $\int_{0^+}^v fg$.

12.2.1 Improper integral of a product

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

Lemma 29. *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 $\int_u^{+\infty} g$ exists. Then $\int_u^{+\infty} fg$ exists, and*

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

Proof. Since f is bounded on $[u; +\infty)$, let $[m; M] := \mathrm{hull}\{f(t) \mid t \geq u\}$. Suppose without loss of generality that $g \geq 0$ on $[u; +\infty)$. Let $v \geq u$. For $u \leq t \leq v$, we have $m \cdot g(t) \leq f(t) \cdot g(t) \leq M \cdot g(t)$, hence $m \cdot \int_u^v g \leq \int_u^v fg \leq M \cdot \int_u^v g$. Let $\varepsilon > 0$. Since g is integrable, the Cauchy criterion gives some neighborhood P of $+\infty$ such that $\forall u, v \in P$, $|\int_u^v g| < \frac{\varepsilon}{1+\max(|m|,|M|)}$. But $|\int_u^v fg| \leq \max(|m|, |M|) \cdot \int_u^{+\infty} g < \varepsilon$; hence fg is integrable. Moreover $m\int_u^{+\infty} g \leq \int_u^{+\infty} fg \leq M\int_u^{+\infty} g$. Thus $\int_u^{+\infty} fg \in [m; M] \cdot \int_u^{+\infty} g$. If $g \leq 0$, the proof is similar. □

We provide an effective version of the previous lemma, in the same spirit as Lemma 26, with a similar proof:

Lemma 30. *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 $u \in \mathbf{u}$:*

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

12.2.2 Catalog of supported integrable functions

In order to be able to use Lemma 30, we need to be able to find a suitable extension $I_g : \mathbb{I} \to \mathbb{I}$ for the improper part of the integral of g. We thus look at two scales of well-known integrable functions for which we can build I_g: Bertrand functions (Section 12.2.2) and exponential functions (Section 12.2.2).

Bertrand integrals

We consider functions $g(x) = x^\alpha \ln^\beta x$ with $\alpha \in \mathbb{R}, \beta \in \mathbb{R}$, and which from now on we will call *Bertrand* functions. These functions are of constant positive sign on $[1; +\infty)$. They are integrable at $+\infty$ only when $\alpha < -1$, or when $\alpha = -1$ and $\beta < -1$. Now we focus on how to compute them. If $\alpha < -1$, $\beta = 0$ and $u > 0$,

$$\int_u^{+\infty} x^\alpha \, dx = -\frac{u^{\alpha+1}}{\alpha+1}. \tag{12.13}$$

When $\beta \geq 1$, integrating by parts shows that

$$\int_u^{+\infty} x^\alpha \ln^\beta x \, dx = -\left(\frac{u^{\alpha+1} \ln^\beta u}{\alpha+1}\right) - \frac{\beta}{\alpha+1} \int_u^{+\infty} x^\alpha \ln^{\beta-1} x \, dx. \tag{12.14}$$

Note that in order to prove this identity, we had to extend Coquelicot with a proof of the general formula for integration by parts (more details in Section 12.2.5).

When $\alpha < -1$ and $\beta < 0$, there is no closed form, but by moving $\ln^\beta x$ into the bounded part of Lemma 29, we can nevertheless compute bounds on the integral. If f is bounded on $[u; \infty]$, then

$$\int_u^{+\infty} f \cdot x^\alpha \ln^\beta x \, dx = \int_u^{+\infty} \underbrace{\left(f \ln^\beta x\right)}_{\text{bounded}} \cdot x^\alpha \, dx$$

and then Formula (12.13) can be used in Lemma 29.

When $\alpha = -1$ and $\beta < -1$, we have a closed form:

$$\int_u^{+\infty} \frac{\ln^\beta x}{x} \, dx = -\frac{\ln^{\beta+1} u}{\beta+1}.$$

When $\beta \in \mathbb{N}$, using Equations (12.13) and (12.14), we can recursively obtain a closed form to evaluate Bertrand integrals with $\alpha < -1$ and $\beta \geq 0$. For instance, using (12.14) then (12.13), we get:

$$\int_1^{+\infty} \frac{\ln x}{x^2} \, dx = -\left(\frac{1^{-1} \ln 1}{-1}\right) - \frac{1}{-1} \int_1^{+\infty} \frac{dx}{x^2} = 0 + (-)\frac{1^{-1}}{-1} = 1.$$

Exponential

We also handle the case of the positive function $g(x) = e^{\gamma x}$ with $\gamma < 0$, using the fact that

$$\int_u^{+\infty} e^{\gamma x} \, dx = -\frac{e^{\gamma u}}{\gamma}.$$

12.2.3 Case of 0^+

When the singular bound is 0^+ instead of $+\infty$, we use a variant of Lemma 29.

Lemma 31. *Let $f, g : \mathbb{R} \to \mathbb{R}$. Suppose that, on $(0; v]$, f is bounded, f and g are continuous, and g has a constant sign. Moreover, suppose that $\int_{0^+}^{v} g$ exists. Then $\int_{0^+}^{v} fg$ exists, and*

$$\int_{0^+}^{v} fg \in \mathrm{hull}\{f(t) \mid 0 \le t \le v\} \cdot \int_{0^+}^{v} g.$$

As in the case of $+\infty$, we provide a catalog of supported functions. Consider $g(t) = t^\alpha(-\ln t)^\beta$ with $\alpha \in \mathbb{R}, \beta \in \mathbb{R}$. This function is of constant sign on $(0; v]$, where $v < 1$. Observe that using the substitution $t = \frac{1}{x}$, we get

$$\int_{0^+}^{v} t^\alpha(-\ln(t))^\beta dt = \int_{1/v}^{\infty} x^{-2-\alpha} \ln^\beta x \, dx.$$

The right-hand-side integral has the shape treated in Section 12.2.2, so we have a way to bound the left-hand-side integral. To do so, we added a proof of the substitution lemma to Coquelicot (see Section 12.2.5).

Note that the functions recognized by the CoqInterval library are actually of the shape $t^\alpha \ln^\beta t$, with α and β integers.

12.2.4 Integrability

Just like in the case of proper integrals in Section 12.1.6, we deduce integrability from computation.

As mentioned above, for improper integrals $\int_u^v fg$, the enclosure of the function f has to be bounded. This is not always very easy to obtain in the case of an integration domain extending to $+\infty$. Indeed, the input domain $\vec{\mathbf{x}}$ is no longer bounded in that case, which means that RPAs become useless. Let us consider the following integral to illustrate the issue ($u > 0$):

$$\int_u^{+\infty} \frac{x+1}{x+2} e^{-x} \, dx.$$

The quotient is bounded on $[u; +\infty)$. Yet using naive interval arithmetic gives $[u+1; +\infty)/[u+2; +\infty) = [0; +\infty)$, which is not bounded. Thus the tactic is unable to prove integrability and to compute an enclosure of the integral. To circumvent this issue, the user has to massage the bounded part of the integrand into a form suitable for naive interval arithmetic, *e.g.* $(1+1/x)/(1+2/x)$. This time, the tactic obtains $[1/(1+2/u); 1+1/u]$, which is bounded. Note that this kind of transformation of the integrand is not always possible. Consider for example the integral

$$\int_1^{\infty} \frac{\ln x}{x} e^{-x} \, dx$$

One would hope to apply Lemma 29 by proving that $f(x) := \frac{\ln x}{x}$ is bounded on $[1; \infty]$ and using e^{-x} as the function g. However, given the primitives of $\mathcal{E}$ there is no way to massage the expression $\frac{\ln x}{x}$ so that naive interval arithmetic

sees is as bounded. In fact, we can not compute an enclosure of this integral automatically as of now.

We conclude this part about integrability with a remark about the fact that the only functions integrable at ∞ that we handle are bounded, whereas of course there exist unbounded integrable functions. One might be tempted to think that the only functions in the class $\mathcal{E}$ which are integrable at $+\infty$ are necessarily bounded. This is actually false, as the following example suggested by Philippe Dumas shows:

$$\int_0^{\infty} t \left(\frac{1+\sin^2 t}{2}\right)^{t^5} dt < \infty.$$

12.2.5 Extending Coquelicot

In order to be able to handle improper integrals, a nontrivial amount of work was needed to improve the existing results provided by the Coquelicot library. This section dives into more technical details about implementations of concepts in Coq. Although some parts of this section may seem at times to be overly focused on proof-engineering details, these reflect the kind of challenges which arise when one wants to extend a library of proofs and tactics. First, we will look at the way the Coq standard library defines real numbers and the basic notions of Real Analysis. Then, we will examine the Coquelicot library which extends the standard library and improves it usability. Finally, we will explain the work which was necessary to extend the tactic to improper integrals.

The Coq standard library of real numbers

Coq's standard library `Reals`[1] axiomatizes real arithmetic with a classical flavor [92]. This means that a type `R` of real numbers is declared as an axiom (as opposed to being constructed) and that the elements 0, 1, the operators $+,*,/,\cdot^{-1}$ are assumed as well as a list of axioms. For example, the comparison operator $<$ is assumed to be total:

```
Axiom total_order_T : ∀ r1 r2:R, {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

This means that given two reals x and y, it is *decidable* to compare them, i.e. there exists a Coq function whose result determines either $x < y$, $x = y$, or $x > y$. This is a very strong assumption since in practice, for any given concrete representation of real numbers, such a function *can't be built.* The library also contains the following axiom:

```
Axiom completeness :
∀ E : R -> Prop, bound E -> (∃ x : R, E x) -> {m : R | is_lub E m}
```

where `is_lub` is defined in the following way:

```
Definition is_upper_bound (E:R -> Prop) (m:R) := ∀ x:R, E x -> x <= m.

Definition is_lub (E:R -> Prop) (m:R) :=
is_upper_bound E m ∧  (∀ b:R, is_upper_bound E b -> m <= b).
```

The `completeness` lemma states that any bounded non-empty subset of $\mathbb{R}$ has a computable least upper-bound. Note that this is both very strong (it

[1] https://coq.inria.fr/distrib/current/stdlib/

postulates a Coq program which takes as input any bounded non-empty subset of $\mathbb{R}$ and returns its upper-bound u) and weaker than what is typically assumed in textbook mathematics: this axiom is not enough to obtain a sequence $(u_n)_{n\in\mathbb{N}}$ of elements of E converging to u.

The standard library comes with a formalization of the properties of usual mathematical functions like sin, cos, exp, and so on, defined as series. It also provides some notions of elementary real analysis, including the definition of continuity, differentiability, and Riemann integrability:

```
Definition RiemannInt (f:R -> R) (u v:R) (pr:Riemann_integrable f u v) : R.
```

This definition uses a dependent type: one needs a proof `pr` that f is Riemann-integrable between u and v in order to write the object $\int_u^v f$. If one wants to add two integrals $\int_u^v f$ and $\int_u^v g$, one first needs to come up with a proof that $f+g$ is integrable between u and v before one can even *write down* the integral $\int_u^v f+g$.

Coquelicot

This section describes the Coquelicot library as it stood before the present work. This library is a conservative extension of the theory of real numbers from the Coq standard library, meaning that it uses the same axiomatization of real numbers and does not add any axiom of its own.

Coquelicot introduces the notion of *filters* in order to manipulate limits in a uniform (e.g. between sequences, functions and functionals) and elegant (with less δ/ε manipulations) way; this idea is borrowed from a similar scheme in Isabelle/HOL [74]; the notion dates back to Cartan [23] [21]. A filter is a family of sets with properties related to inclusion (see Definition 16). Filters allow to treat in similar ways various kinds of limits such as sequence limits, limits of functions at one point or on one side of a point, and asymptotic limits. They can in fact be thought of as a way to encode the familiar notion of neighborhoods in analysis.

Definition 16 (filter). *Let E be a set. $F \in \mathcal{P}(E)$ is a filter of E if:*

- *$E \in F$;*
- *For all A, $B \subset E$, if $A \in F$ and $B \in F$ then $A \cap B \in F$;*
- *If $A \in F$ and $A \subset B$, then $B \in F$.*

Here is an example of filter containing the neighbourhoods of ∞:

Definition 17. *Let $f_\infty := \{E \subset \mathbb{R} \,|\, \exists M \in \mathbb{R}, \forall x \geq M, x \in E\}$. Then f_∞ is a filter of $\mathbb{R}$.*

Coquelicot has an inductive type `Rbar` encoding reals extended with ∞ and $-\infty$:

```
Inductive Rbar : Set :=
| Finite  : R -> Rbar
| p_infty : Rbar
| m_infty : Rbar.
```

A filter encoding a neighborhood of `x` $\in$ `Rbar` can be obtained by evaluating `Rbar_locally x`:

```
Definition Rbar_locally (a : Rbar) (P : R -> Prop) :=
match a with
| Finite a => locally a P
| p_infty => ∃ M : R, ∀ x, M < x -> P x
| m_infty => ∃ M : R, ∀ x, x < M -> P x
end.
```

The predicate `locally a P` expresses that the predicate P is valid in a neighborhood of the real number a.

Given a point `x` $\in$ `R`, `at_point x` is a filter for `x`:

```
Definition at_point (P : R -> Prop) : Prop := P x.
```

Coquelicot also features a formalization of the product of two filters:

Lemma 32 (product of two filters)**.** *Let F be a filter of the set S and G be a filter of the set T. Suppose $Q \in F$ and $R \in G$. Then $\{P \in S \times T \,|\, Q \times R \subset P\}$ is a filter of $E \times F$.*

Coquelicot adopts a systematic approach of replacing dependent types used in the standard library (like in Listing 12.2.5) with total functions. It also provides an alternative formalization of the theory of Riemann integration. Let $\mathbb{V}$ be a complete normed $\mathbb{R}$-module:

```
Variable V : CompleteNormedModule R.
```

The definition of the (proper) Riemann integral between two reals a and b has the type:

```
Definition is_RInt (f : R -> V) (a b : R) (If : V) : Prop.
```

and it expresses the fact that Riemann sums of f between a and b converge to the value `If` $\in \mathbb{R}$ when their step tends to 0. Coquelicot provides a *total* operator that outputs a value in $\mathbb{R}$ from a function $f : \mathbb{R} \to \mathbb{R}$ and two bounds $u, v \in \mathbb{R}$:

```
Definition RInt (f : R -> V) (u : R) (v : R) : V.
```

When the function f is Riemann-integrable on $[u; v]$, the value `(RInt f u v)` is equal to $\int_u^v f(t)\,dt$. Otherwise it is left unspecified and thus most properties about the actual value of `(RInt f u v)` hold only if f is integrable on $[u; v]$.

The library also features a notion of generalized Riemann integrals used to represent integrals such as $\int_{0^+}^{+\infty} \ln x/(1+x^2)dx$. The bounds of such integrals are not real numbers, but filters. There is a predicate `is_RInt_gen` similar to `is_RInt`:

```
Definition is_RInt_gen :
(R -> V) -> ((R -> Prop) -> Prop) -> ((R -> Prop) -> Prop) -> V -> Prop.
```

The definition of `is_RInt_gen` amounts to the following. Let $f : \mathbb{R} \to \mathbb{V}$ be a function, and let u, v be two bounds. If $l \in \mathbb{R}$, l is said to be the generalized integral of f between u and v if $\int_a^b f(t)dt$ is well-defined for all (a, b) in a neighborhood of (u, v) and if $I_f(a, b) \to l$ when $(a, b) \to (u, v)$.

Contrary to the previous case of proper integrals, the library does not feature a total function computing the improper Riemann integral of f between two filters F and G. The function `is_RInt_gen` is in fact a *relation*. Even with a proof that this relation locally describes a function, it is not possible to directly mention the value of this function, i.e. the integral. For example, we can write in Coquelicot

```
is_RInt_gen (fun x => exp(-x²)) (at_point 0) (Rbar_locally p_infty) √π/2
```

to mean that $\int_0^\infty e^{-x^2}dx = \frac{\sqrt{\pi}}{2}$ but it is not as straightforward to express that $\int_0^\infty e^{-x^2}dx \in [0;1]$:

```
∃ l, is_RInt_gen (fun x => exp(-x²)) (at_point 0) (Rbar_locally p_infty) l ∧
    l ∈ [0;1].
```

Using such statements would have made it quite cumbersome for us to extend our tactic to generalized integrals.

New definitions

In order to be able to state and prove in Coq the theorems of Sections 12.2.1, 12.2.2, 12.2.3 and 12.2.4 as well as to extend the tactic to improper integrals, we needed a total operator computing such integrals and some more lemmas of Real Analysis on them.

A new definition of `is_RInt_gen`: We redefined `is_RInt_gen` in a more natural way in order to simplify proofs. If $a \in \mathbb{R}$ and $f : \mathbb{R} \to \mathbb{R}$, a natural way to define $\int_a^\infty f$ is

$$\int_a^\infty f = \lim_{b \to \infty} \int_a^b f,$$

where this equality is meant as: "there exists a neighborhood U of ∞ such that when $b \in U$, $\int_a^b f$ exists and $\lim_{b \in U, b \to \infty} \int_a^b f$ exists". This is so cumbersome to express because the function $(f, a, b) \mapsto \int_a^b f$ itself is partial. Since the generalized integral is in some sense the "limit" of this relation, this motivated the introduction of the notion of limit of a partial function defined implicitly by a predicate:

```
Definition filterlimi {T U : Type} (R : T -> U -> Prop) F G : Prop.
```

For any relation R on elements x of type `T` and y of type `U`, and for any filters `F` and `G`, the predicate `filterlimi` R `F G` states that for any neighborhood $Q \in G$, the set $\{x \in T | \exists y \in Q,\ R\,x\,y\}$ of "antecedents" of Q by R is itself a neighborhood of F. Now we are ready to read the definition of `is_RInt_gen`:

```
Definition is_RInt_gen (f : R -> V) (Fa Fb : (R -> Prop) -> Prop) (l : V) :=
filterlimi
(fun (a,b) => is_RInt f a b)
(filter_prod Fa Fb)
(locally l).
```

This is saying that for any function $f : \mathbb{R} \to \mathbb{V}$, for any real numbers a and b, for any neighborhood `W` of `l`, for any `w` ∈ `W`, the set of pairs `(a,b)` such that `is_RInt f a b w` holds is in the filter product of `Fa` and `Fb`.

A Total Operator for Generalized Integrals: The following non-computable `iota` function is useful for building a total function from a predicate on a complete space `T`:

```
Definition iota (P : T -> Prop) : T := lim (fun A => (∀ x, P x -> A x)).
```

The function `lim` is here given as input a filter `N` formed of all sets `A` containing `P`. Assuming 1) that `N` does not contain the empty set and 2) that for all ε, there is some x such that the ball $\mathcal{B}(x, \varepsilon) \subseteq$ `N`, `lim N` returns an element `x` $\in$ `T` such that all open balls centered on `x` are in `N`. In the particular case above, if one can prove that any two points in `P` are equal (which is the case for `is_RInt` or `is_RInt_gen`, for instance), `iota` is effectively a choice function for the input property. If the property `P` is true for exactly one value x of the complete space `T`, then `iota P` returns this value; otherwise, it returns a default value.

Using `iota`, we defined the function

```
Definition RInt_gen (f : R -> V) (a b : (R -> Prop) -> Prop) :=
iota (is_RInt_gen f a b).
```

Just like for `RInt`, the output of this function only makes sense if `f` is integrable between the bounds `a` and `b`. Notice that `RInt_gen` is more general than `RInt`: it takes as argument a function whose values are in a complete normed $\mathbb{R}$-module. We redefined `RInt` in a similar way (defined on more general functions):

```
Definition RInt (f : R -> V) (a b : R) :=
iota (is_RInt f a b).
```

Unanticipated Consequences of the New Definitions

As we saw above, the new definition of `RInt` using `iota` accepts functions with a more general type. This allows for example to define proper integrals of complex-valued functions. As we defined it in this more general way, we realized that some elementary theorems about `RInt` had been proven using arguments which are specific to $\mathbb{R}$. For instance,

```
Lemma is_RInt_derive (f df : R -> V) (a b : R) :
(∀ x : R, Rmin a b <= x <= Rmax a b -> is_derive f x (df x)) ->
(∀ x : R, Rmin a b <= x <= Rmax a b -> continuous df x) ->
is_RInt df a b (minus (f b) (f a)).
```

which states that if f is continuously differentiable on $[a, b]$, and if $df : \mathbb{R} \to \mathbb{R}$ coincides with f' on $[a, b]$, then

$$\int_a^b df(x)dx = f(b) - f(a), \tag{12.15}$$

had been proved for $f : \mathbb{R} \to \mathbb{R}$ using the Intermediate Value Theorem and Riemann sums. Such a proof does not generalize to normed modules in general. A more usual way of proving this is to set

$$F(x) := f(a) + \int_a^x df(x)\, dx - f(x) \tag{12.16}$$

and then to observe that

$$\forall x \in [a, b], F'(x) = 0 \tag{12.17}$$

and that $F(a) = 0$, so that $\forall x \in [a, b], F(x) = 0$ hence the result. However, it turns out that the following lemma, implicitly used in this usual proof, was not proved in Coquelicot with the necessary generality:

```
Lemma eq_is_derive :
∀ (f : R -> V) (a b : R),
(∀ t, a <= t <= b -> is_derive f t zero) ->
a < b -> f a = f b.
```

This lemma states that if the derivative of a function f is uniformly zero in $[a; b]$, then f is a constant function. It is in fact not so trivial as it may seem at first, because the first simple proof which comes to mind relies on the theorem `is_RInt_derive` which we are trying to prove. Our proof relies on the `completeness` axiom of the `Reals` library presented in Section 12.2.5 which states the existence of a least upper bound for bounded and non-empty subsets of $\mathbb{R}$. Here is our proof of `eq_is_derive`:

Proof. Let $\varepsilon > 0$. We want to prove that for all $x \in [a, b]$, $|f(x) - f(a)| \leq \varepsilon$. Let $A = \{x \leq b \mid \forall t, a \leq t \leq x \implies |f(t) - f(a)| \leq \varepsilon(t - a)\}$. A is non-empty since $a \in A$, and A is certainly bounded by b, so that A has an upper bound u. Let $c \in A$ be such that $c < b$.

Since $f'(c) = 0$, we can find $\delta > 0$ so that $c + \delta \leq b$ and

$$\forall x \in [c - \delta; c + \delta], f(x) - f(c) \leq (x - c)\varepsilon$$

If $x \in [c; c + \delta]$, since $c \in A$,

$$\begin{aligned} |f(x) - f(a)| &\leq |f(x) - f(c)| + |f(c) - f(a)| \\ &\leq (x - c)\varepsilon + (c - a)\varepsilon \\ &= (x - a)\varepsilon. \end{aligned}$$

Thus, $x \in A$ so that $[c; c + \delta] \subset A$.

We know that $u \leq b$. Moreover, u cannot be less than b by the above argument. The only remaining option is that $b = u$. Hence, for all $x \in [a; b]$, $|f(x) - f(a)| \leq \varepsilon$. Since this is true for arbitrary ε, we conclude that

$$\forall x \in [a; b], f(x) = f(a).$$

□

In order to finish the proof of `is_RInt_derive`, we also needed a proof of the fact that the derivative of $\int_a^x f(x)$ at $x \in [a; b]$ is $f(x)$, which in Coquelicot is formulated more generally as:

```
Lemma is_derive_RInt (f If : R -> V) (a b : R)
(Hloc : locally b (fun x => is_RInt f a x (If x))) :
-> continuous f b
-> is_derive If b (f b).
```

Listing 12.1: Derivative of an integral in Coquelicot

The first hypothesis `Hloc` is very strong. Applied to our specific case, and assuming $a \leq b$, it mandates that $\int_a^x f(x)dx$ exist in a *neighborhood* of b, that is a little to the left of b (that we can do with, especially when $a < b$) but also to the right of b. This has no chance to be true in our case, as the following counterexample shows: Take $a = -1$, $b = 0$, and $f(x) = x$ on $[-\infty, 0]$, and $f(x) = x\chi_{\mathbb{Q}}(x)$ on $[0, \infty]$, where $\chi_{\mathbb{Q}}$ is the characteristic function of $\mathbb{Q}$ on $\mathbb{R}$, i.e $\chi_{\mathbb{Q}}$ returns 1 on rational numbers and 0 on irrational numbers. Then f satisfies all the preconditions of `is_RInt_derive`, as it is $\mathcal{C}^1$ at all points of $[a, b] = [-1, 0]$:

- f is $\mathcal{C}^1$ trivially at any point in $[-1, 0)$;
- f is continuous at any point of $[-1, 0]$.
- f' is defined and is continuous to the left of 0, with $f'_{\text{left}}(0) = 1$;
- For $x \geq 0$, $f(x) - f(0) - 1 \cdot x = x(\chi_{\mathbb{Q}}(x) - 1) \to 0$ when $x \to 0$, hence $f'_{\text{right}}(0) = 1$ and thus $f'(0) = 1$.

However it can't be that $f(x) = \int_a^x df(x)dx$ in a neighborhood of 0, as the quantity on the right is continuous on that neighborhood while f is not, at least to the right of 0.

Therefore, how can we use `is_derive_RInt` in the proof of `is_RInt_derive`? We want

$$\forall x \in [a; b], \left(\int_a^x df(x)dx\right)' = df(x) \tag{12.18}$$

However, in the hypotheses of `is_RInt_derive`, the function f is continuously differentiable on $[a; b]$, so that df is continuous, which is weaker than what the `Hloc` hypothesis of `is_derive_RInt` mandates. Let g be a $\mathcal{C}^1$ extension of f on $[a; b]$, meaning that f and g coincide on $[a; b]$, and g is affine on $(-\infty; a]$ and on $[b; +\infty)$, and $\mathcal{C}^1$ on $\mathbb{R}$. Then since g' is continuous everywhere, $x \mapsto \int_a^x g'(x)dx$ is defined everywhere so that `Hloc` is true for $x \mapsto \int_a^x g'(x)dx$ and g'. Hence `is_derive_RInt` tells us that

$$\forall x \in [a; b], \left(\int_a^x g'(x)dx\right)' = g'(x) \tag{12.19}$$

But g' and df coincide on $[a; b]$, which entails (12.18).

Substitution lemma

We also (re-)proved the substitution lemma for proper integrals for the same reason as `is_RInt_derive`: the proof was using the Intermediate Value Theorem which is specific to real-valued functions and was broken by the generalization of the type of `is_RInt`. This lemma was crucial for handling Bertrand integrals at 0^+ (see Section 12.2.3).

```
Lemma is_RInt_comp (f : R -> V) (g dg : R -> R) (a b : R) :
(∀ x, Rmin a b <= x <= Rmax a b -> continuous f (g x)) ->
(∀ x, Rmin a b <= x <= Rmax a b -> is_derive g x (dg x) ∧  continuous dg x) ->
is_RInt (fun y => scal (dg y) (f (g y))) a b (RInt f (g a) (g b)).
```

Just like in Section 12.2.5, this proof uses a continuously differentiable extension of the function g.

12.2.6 Conclusion

The changes described in this section are now part of Coquelicot `3.0.0` and later versions.

12.3 Benchmarks

This section presents the behavior of the tactic on several integration problems, each given as a symbolic integral, its value (approximate if no closed form exists), and a set of absolute error bounds that must be reached by the tactic. Each problem is translated into a set of Coq scripts as follows, one for each bound:

```
Goal Rabs (RInt[_gen] function domain - value) <= error.
interval with options.
Qed.
```

The tactic options have been set using the following experimental protocol. First, the target relative accuracy is computed from the error bound and the initial estimation of an integral. The floating-point precision is then set at about 10 more bits than the target accuracy, so that round-off errors do not make interval enclosures too large. The maximal depth is originally set to a large enough value. Then, various degrees of RPAs are tested and the one that leads to the fastest execution is kept. Finally, the maximal depth is reduced as long as the tactic succeeds in proving the bounds, so that we get an idea of how deep splitting has to be performed to compute an accurate enclosure of the integral. Note that reducing the maximal depth might improve timings in case the adaptive algorithm had been overly conservative and did too much domain splitting. Reducing the target relative accuracy could also improve timings (again by preventing some domain splitting), but this was not done.

The tables below indicate, for each error bound, the time needed and the tactic settings. Timings are in seconds and are obtained on a standard-grade laptop. All timings are obtained with a version of the tactic that uses the `vm_compute` machinery to perform computations. Modern versions of Coq also have a `native_compute` machinery [18], which can improve the longest computation times by an order of magnitude, at the expense of a long initialization time which makes it unsuitable for the present benchmarks.

12.3.1 Proper integrals

For each proper integral, we also ran several quadrature methods from Octave [40]: `quad`, `quadv`, `quadgk`, `quadl`, `quadcc`. We also used IntLab [109]; it provides `verifyquad`, an interval arithmetic procedure that computes integral enclosures using a verified Romberg method. For each method, we ask for an absolute accuracy of 10^{-15}. We only comment when the answer is off, or when the execution time exceeds 1 second. Finally, we also tested VNODE-LP [98] on each example by representing the integral as the value of the solution of a differential equation.

The first problem is the integral of the derivative of arctan, a highly regular function. As expected, the tactic behaves well on it, since it takes about 3 seconds to compute 18 decimal digits of π by integration. Note that the time needed for reifying the goal and performing the initial computations is incompressible, so there is not much difference between 10^{-3} and 10^{-6}.

$$\int_0^1 \frac{dx}{1+x^2} = \frac{\pi}{4}$$

Error	Time	Accy	Degree	Depth	Prec
10^{-3}	0.3	10	15	0	30
10^{-6}	0.3	20	6	2	30
10^{-9}	0.6	30	7	3	40
10^{-12}	1.0	40	7	4	50
10^{-15}	1.7	50	10	5	60
10^{-18}	2.9	60	12	5	70

The second problem is Ahmed's integral [3]. It is a bit less regular and uses more operators than the previous problem, but the tactic still behaves well enough: adding ten bits of accuracy doubles the computation time.

$$\int_0^1 \frac{\arctan \sqrt{x^2+2}}{\sqrt{x^2+2}\,(x^2+1)} dx = \frac{5\pi^2}{96}$$

Error	Time	Accy	Degree	Depth	Prec
10^{-3}	0.5	9	5	1	30
10^{-6}	1.2	19	7	3	30
10^{-9}	2.8	29	7	3	40
10^{-12}	5.5	39	10	3	50
10^{-15}	11.2	49	10	4	55

The third problem involves a function that is harder to approximate using RPAs, so the tactic performs more domain splitting, degrading performances.

$$\int_0^\pi \frac{x \sin x}{1+\cos^2 x} dx = \frac{\pi^2}{4}$$

Error	Time	Accy	Degree	Depth	Prec
10^{-3}	1.1	11	9	2	30
10^{-6}	2.3	21	6	5	30
10^{-9}	5.0	31	9	5	40
10^{-12}	11.5	41	11	7	50
10^{-15}	27.2	51	11	7	65

The fourth problem is an example from Helfgott [69] in the spirit of [70]. The polynomial part crosses zero, so there is a point where the integrand is not differentiable because of the absolute value. Thus only degenerate Taylor models can be computed around that point. Although the tactic has to perform a lot of domain splitting to isolate that point, it still computes an enclosure of the integral quickly. Note that the approximate value of the integral was computed using the `interval_intro` tactic.

$$\int_0^1 \left|\left(x^4+10x^3+19x^2-6x-6\right) e^x\right| dx \simeq 11.14731055005714$$

On this example, quadrature methods have some troubles: `quad` gives only 10 correct digits; `verifyquad` gives a false answer (a tight interval not containing the value of the integral) without warning;[2] `quadgk` gives only 9 correct digits. VNODE-LP cannot be used because of the absolute value.

Error	Time	Accuracy	Degree	Depth	Precision
10^{-3}	0.7	14	5	8	30
10^{-6}	0.9	24	6	13	40
10^{-9}	1.3	34	8	18	50
10^{-12}	1.9	44	10	22	60
10^{-15}	2.7	54	12	28	70

[2]The bug lies in an incorrect implementation of Taylor models for absolute value. It has now been fixed by removing support for absolute values.

The last two problems are inherently hard to numerically integrate. The first one is the 12-th coefficient of a Chebyshev expansion. Note that the initial estimation of the integral is completely off, which explains why the relative accuracy has to be set about 30 bits higher than one would expect. As with the previous problem, there are some points where no RPAs can be computed. The approximate value was again computed using the `interval_intro` tactic.

$$\int_{-1}^{1} \left(2048x^{12} - 6144x^{10} + 6912x^{8} - 3584x^{6} + 840x^{4} - 72x^{2} + 1\right) \exp\left(-\left(x - \tfrac{3}{4}\right)^{2}\right) \sqrt{1 - x^{2}}\, dx \simeq -3.2555895745 \cdot 10^{-6}$$

The `quad`, `quadl`, and `quadcc` procedures give completely off but consistent answers without warning; `quadv` gives an answer which is off the mark as well, but it gives a warning "maximum iteration count reached"; `verifyquad` works only for functions that are four times differentiable, hence its failure here; `quadgk` gives yet another off answer with no warning. Finally, VNODE-LP fails here because of computational errors such as divisions by 0.

Error	Time	Accuracy	Degree	Depth	Precision
10^{-6}	10.7	32	8	17	40
10^{-9}	22.9	42	10	22	50
10^{-12}	48.3	52	13	28	60
10^{-15}	111.8	62	13	35	70

The last problem is an example taken from Tucker's book [121] and originally suggested by Rump in [109, page 372]. This integral is often incorrectly approximated by computer algebra systems, because of the large number of oscillations (about 950 sign changes) and the large value of the n-th derivatives of the function. While the maximal depth is not too large, the tactic reaches it for numerous subdomains, hence the large computation time.

The `quad`, `quadcc`, and `quadgk` procedures give off values without any warning; `quadv` gives an off value with a warning; `verifyquad` takes 1.7 seconds to give a correct answer; `quadl` takes 9 seconds to return a correct answer.

$$\int_{0}^{8} \sin(x + e^{x})\, dx \simeq 0.3474$$

Error	Time	Accy	Degree	Depth	Prec
10^{-1}	81.0	6	6	12	30
10^{-2}	123.6	9	8	12	30
10^{-3}	183.4	12	10	12	30
10^{-4}	277.6	15	12	12	30

12.3.2 Improper integrals

Few tools are able to handle unbounded integration domains and even fewer can give reliable bounds on the integral value. So this section is mostly about CoqInterval. The first example shows a simple integrand with an exponential bound:

$$\int_1^{+\infty} \frac{e^{-x}}{\sqrt{x}}\,dx = \sqrt{\pi}\cdot\operatorname{erfc}(1)$$

Error	Time	Accy	Degree	Depth	Prec
10^{-3}	0.4	8	7	3	30
10^{-6}	0.9	18	7	6	30
10^{-9}	2.8	28	9	7	40
10^{-12}	5.6	38	13	7	50
10^{-15}	13.1	48	13	8	60

The second example is similar to the integral from Tucker's book, in the sense that the high number of oscillations of the integrand makes it hard to give an accurate approximation of the integral. For instance, Maple 18 forfeits after 10 seconds of computations. The tactic does not perform much better since it is not able to compute more than two digits in a reasonable amount of time. This is partly due to the adaptive splitting algorithm, which is built upon the assumption that splitting an integration domain into two parts eventually improves the accuracy by more than one bit on each part; this is not the case for the remainder of this example.

$$\int_1^{+\infty} \cos x\,\frac{\ln x}{x^2}\,dx \simeq -0.1595$$

Error	Time	Accy	Degree	Depth	Prec
10^{-1}	2.7	3	6	12	30
10^{-2}	42.6	6	8	19	30

The last example comes from Helfgott's proof of the ternary Goldbach conjecture [70, page 35]:

$$\int_{-\infty}^{\infty} \frac{(0.5\cdot\ln(\tau^2+2.25)+4.1396+\ln\pi)^2}{0.25+\tau^2}\,d\tau$$

The tactic cannot handle this integral fully automatically since the integrand is not syntactically a product with a term $x^\alpha \ln^\beta x$. It is up to the user to split the integral into two parts: one proper part between $-100{,}000$ and $100{,}000$ (as was done in the original paper) and one improper part between $100{,}000$ and $+\infty$ (counted twice, since the integrand is an even function). The proper part is handled in the same way as all the previous examples. It takes about 30 seconds to get the relative accuracy of 10^{-6} needed by the original paper. For the improper part, the integrand first has to be transformed into the following form, which was proved to be equal to the original integrand in a few lines of Coq:

$$\int_{100\,000}^{+\infty} \frac{1+\left(\frac{0.5\cdot\ln(1+2.25/\tau^2)+4.1396+\ln\pi}{\ln\tau}\right)^2}{1+0.25/\tau^2}\cdot\frac{\ln^2\tau}{\tau^2}\,d\tau \simeq 3.17742\cdot 10^{-3}.$$

Error	Time	Accuracy	Degree	Depth	Precision
10^{-3}	0.6	2	3	0	30
10^{-4}	0.8	5	5	2	30
10^{-5}	1.5	8	7	6	30
10^{-6}	3.1	11	9	11	30
10^{-7}	5.6	15	12	12	30
10^{-8}	11.2	18	15	15	30

Bounding the remainder with a low accuracy is sufficient to prove that the integral on the whole domain is included in $[226.849; 226.850]$ and thus that the upper bound 226.844 used in [70] is incorrect, although this does not invalidate the proof. Here we present benchmarks from both the proper and the improper case for estimating integrals.

Chapter 13

Quadrature and Automatic Differentiation of order n

13.1 Quadrature Methods: A quick introduction

As mentioned in Chapter 10, quadrature methods based on the evaluation of a linear combination of evaluations of the integrand f can be made rigorous under certain assumptions on the regularity of f. In this case, an error bound depending on some iterated derivative of the integrand on the integration interval is typically provided, as in the case of Formula (10.3) for the trapezoidal rule. Another example of a popular and efficient quadrature method is the so-called Simpson's method. Rabinowitz is quoted [36] claiming that "95% of all pratical work in numerical analysis boiled down to applications of Simpson's rule and linear interpolation":

Theorem 7 (Simpson's method). *Let $f \in \mathcal{C}^4([u,v])$. Then there exists $\xi \in [u;v]$ such that*

$$\int_u^v f(x)\,dx - \frac{b-a}{6}\left[f(u) + 4f\left(\frac{u+v}{2}\right) + f(v)\right] = -\frac{(v-u)^5}{2880} f^{(4)}(\xi).$$

In practice, the *compound rule* is often used:

Theorem 8 (Compound rule version of Simpson's method). *Let $f \in \mathcal{C}^4([u,v])$ and $n \in \mathbb{N}$, with $n \geq 1$ and $N = 2n$. Let $x_0 = u < x_1 < \cdots < x_{N-1} < x_N = v$ be equally spaced points, with $h = \frac{v-u}{N} = x_{i+1} - x_i$ for all i. For $0 \leq k \leq N$, let f_k denote $f(x_k)$. There exists $\xi \in [u;v]$ such that*

$$\int_u^v f(x)\,dx - \underbrace{\frac{h}{3}\left[f_0 + 4\sum_{k=1}^{n} f_{2k-1} + 2\sum_{k=1}^{n-1} f_{2k} + f_{2n}\right]}_{I_n(f)} = -\underbrace{\frac{(v-u)^5}{180N^4} f^{(4)}(\xi)}_{e_n(f)}.$$

As claimed above, most popular integration software uses Simpson's rule or other Newton-Cotes-based methods to compute approximations of integrals.

They usually only make use of $I_n(f)$ and compute it using floating-point approximations (as opposed to interval methods) and ignore the error term $e_n(f)$.

In order to make Theorem 8 effective and rigorous, we need two things. The first requirement is to correctly evaluate f at the points x_k, which thanks to the methods exposed in Sections 11.1 and 11.2 we know how to do for at least a class of functions f by building an interval extension $\mathbf{F}$ of f which is correct *by construction*. The second requirement is of a nature which has not been mentioned yet: we need to produce a correct enclosure of $f^{(4)}$ at the point ξ. Since ξ could be any point in $[u; v]$, what we really want is an interval evaluation of $f^{(4)}$ on $[u; v]$. All this suggests an interval version of the theorem:

Theorem 9 (Interval compound rule version of Simpson's method)**.** *Let* $f \in \mathcal{C}^4([u, v])$ *and* $n \in \mathbb{N}$, *with* $n \geq 1$ *and* $N = 2n$. *Let* $\mathbf{u}, \mathbf{v} \in \mathbb{I}$ *with* $u \in \mathbf{u}$, $v \in \mathbf{v}$. *Let* $\mathbf{F}$ *be an interval extension of* f, *and let* $\mathbf{G}$ *be an interval extension of* $f^{(4)}$ *on* $[u; v]$. *For* $0 \leq k \leq 2n$, *pose* $\mathbf{h} := \frac{\mathbf{v}-\mathbf{u}}{2n}$ *and* $\mathbf{x_k} := \mathbf{u} + k \cdot \mathbf{h}$ *and* $\mathbf{F_k} := \mathbf{F}(\mathbf{x_k})$. *Then*

$$\int_u^v f(x)\,dx \in \underbrace{\frac{\mathbf{h}}{3}\left[\mathbf{F_0} + 4\sum_{k=1}^{n} \mathbf{F_{2k-1}} + 2\sum_{k=1}^{n-1} \mathbf{F_{2k}} + \mathbf{F_{2n}}\right]}_{\mathbf{I_n}(\mathbf{F})} - \underbrace{\frac{(\mathbf{v}-\mathbf{u})^5}{180N^4}\mathbf{G}([u; v])}_{\mathbf{E_n}(\mathbf{G})}.$$

In order to use Theorem 9 in an implementation, we need a way to estimate the k-th derivative of a function in $\mathcal{E}$ for $k \geq 2$. The usual technique to obtain numerical approximations of k-th derivatives is called Automatic Differentiation (AD): AD of order k is a numerical method to compute an enclosure of the k-th derivative of a function f. AD of order 1 is already implemented in CoqInterval, but not for higher orders.

Once we obtained results using naive interval arithmetic and RPAs, we compared how far off we are from more traditional quadrature methods. For this, we made several attempts at implementing AD of arbitrary order, which are recounted (Section 13.1.1). Then, a few benchmarks about computation times are presented (Section 13.1.2).

13.1.1 AD of order n

Automatic Differentiation or Algorithmic Differentiation (AD) is a process for computing the numerical derivative of a function. We restrict ourselves to the case of a univariate differentiable real function $f : \mathbb{R} \to \mathbb{R}$ and a real number t_0, and the goal is to simultaneously compute an interval approximation of $f(t_0)$ and $f'(t_0)$. AD distinguishes itself from two other natural ways of approximating derivatives: the first one is to symbolically compute the derivative of the function when it is presented as an expression made of additions, multiplications, divisions, composition and usual mathematical functions; the second one is to try to approximate $f'(x) = \frac{f(x+h)-f(x)}{h}$ for some well-chosen h.

There exists a large literature on the topic of AD. The combination of AD with interval arithmetic is covered in Moore [96] and Tucker [121] among others. Interval AD of order 1, which is described below, was formalized in CoqInterval [91]. To our knowledge, the only other implementation in a formal proof system is in a paper focused on differentiating FORTRAN programs using only basic arithmetic operations [93].

Here we restrict ourselves to a specific flavor of univariate AD which uses operator overloading. The functions $f : \mathbb{R} \to \mathbb{R}$ that we consider belong to the class $\mathcal{E}$ presented in Section 11.1.

AD of order 1

Let us first describe automatic differentiation of order 1. Let $e(x) \in \mathcal{E}$ be an expression, and let f be the function defined by $t \mapsto e(t)$. Suppose we want to compute $f(t_0)$ and $f'(t_0)$ for some $t_0 \in \mathbb{R}$ such that these quantities are well-defined.

The idea is to build the pair $p_e(t_0) = (f(t_0), f'(t_0)) \in \overline{\mathbb{R}}^2$ recursively on the structure of $e(x) \in \mathcal{E}$. The operations are done assuming that f is both defined and differentiable at t_0: when $f(t_0)$ or $f'(t_0)$ is ill-defined, it is replaced by the $\bot_{\mathbb{R}}$ value which then propagates. The following rules are used:

- if $e(x) = c$ where c is a constant of $\mathcal{E}$, then $p_e(t_0) = (c, 0)$;
- if $e(x) = x$, then $p_e(t_0) = (x, 1)$;
- if $e(x) = e_g(x) + e_h(x)$, if $p_{e_g}(t_0) = (v_g, v_{g'})$ and $p_{e_h}(t_0) = (v_h, v_{h'})$ then $p_e(t_0) = (v_g + v_h, v_{g'} + v_{h'})$;
- if $e(x) = e_g(x) \cdot e_h(x)$, if $p_{e_g}(t_0) = (v_g, v_{g'})$ and $p_{e_h}(t_0) = (v_h, v_{h'})$ then $p_e(t_0) = (v_g \cdot v_h, v_g \cdot v_{h'} + v_h \cdot v_{g'})$;
- if $e(x) = \frac{e_g(x)}{e_h(x)}$, then if $p_{e_g}(t_0) = (v_g, v_{g'})$ and $p_{e_h}(t_0) = (v_h, v_{h'})$ then $p_e(t_0) = (\frac{v_g}{v_h}, \frac{v_g \cdot v_{h'} - v_h \cdot v_{g'}}{v_h^2})$;
- if $e(x) = (e_g(x))^{-1}$ and $p_{e_g}(t_0) = (v_g, v_{g'})$, then $p_e(t_0) = (v_g^{-1}, -v_{g'} \left(v_g{}^{-1}\right)^2)$;
- If $e(x) = u_f(g(x))$, where u_f is a usual function (such as $\cdot^n$, exp, log, sin, cos, tan, or any unary function of $\mathcal{E}$) and $g(x) \in \mathcal{E}$, then if $p_{e_g}(t_0) = (v_g, v_{g'})$, $p_e(t_0) = (u_f(v_g), v_{g'} \cdot u_f'(v_g))$;
- If $e(x) = |e_g(x)|$ and $p_{e_g}(t_0) = (v_g, v_{g'})$, then $p_e(t_0) = \begin{cases} (v_g, v_{g'}) & \text{if } v_g > 0 \\ (v_g, -v_{g'}) & \text{if } v_g < 0 \\ \bot_{\mathbb{R}} & \textit{otherwise} \end{cases}$

The following lemma describes the correctness of this process for constructing $p_e(t_0)$:

Lemma 33. *For any expression $e(x) \in \mathcal{E}$, for all $t_0 \in \mathbb{R}$ if $f := t \mapsto e(t)$ is differentiable at t_0, then*

$$p_e(t_0) = (f(t_0), f'(t_0)).$$

Proof. The proof is by induction on the structure of $e(x) \in \mathcal{E}$:

- It is straightforward for the two base cases of c and x;
- For the three binary operators $+$, $\cdot$, and $/$, it simply amounts to the well-known derivation rules for these operators;

- For unary operators corresponding to a usual function u_f, correctness is deduces from the formula for differentiating u_f and the chain rule;
- For the absolute value, it is a simple case analysis.

□

The above description is not yet an effective algorithm, because the data type of the pair $p_e(x)$ is $\overline{\mathbb{R}}^2$. In order to make an implementation which is able to provide correct numerical enclosures of first-order derivatives, we replicate all the above steps with the usual interval operations and, from an input interval $\mathbf{t_0}$, build a pair of intervals $\mathbf{p_e}(\mathbf{t_0})$ with the following property:

Lemma 34. *For any expression $e(x) \in \mathcal{E}$, for all $\mathbf{t_0} \in \mathbb{I}$ and $t_0 \in \mathbb{R}$ such that $t_0 \in \mathbf{t_0}$,*

$$p_e(t_0) \in \mathbf{p_e}(\mathbf{t_0}).$$

Proof. Since $\mathbf{p_e}(\mathbf{t_0})$ is built the same way as $p_e(t_0)$, and since every interval operation used is an interval extension of the corresponding real operation, the result follows by induction on the structure of e. □

The pair $\mathbf{p_e}(\mathbf{t_0}) := (\mathbf{i_0}, \mathbf{i_1})$ also has the nice property that if $\mathbf{i_1} \neq \bot_{\mathbb{I}}$, then $f := t \mapsto e(t)$ is differentiable at t_0. This property is similar to the way integrability is inferred in Section 12.1.6.

This process of interval automatic differentiation of order 1 is not a contribution: it is already implemented in CoqInterval and is used for example to refine some interval enclosures by looking at the sign of the derivative.

AD of Order n

It is natural to want to extend the principle of the previous subsection to the n-th derivative by computing, still by induction on the structure of the expression $e(x)$ defining f, the vector $p_e^n(x) := (f(x), f'(x), \cdots, f^{(n)}(x)) \in \mathbb{R}^{n+1}$.

When implementing order 1 automatic differentiation, the only variation in the derivation formulas which might have an impact on numerical accuracy is the associativity and distributivity of operators.

A typical presentation of AD of order n establishes a direct formula for the n-th derivative of each operator. Up to an integer factor, the elements of $p_e^n(t_0)$ are really the $n+1$ first coefficients of the Taylor series development of f about t_0. Formulas for each operation in the definition of $\mathcal{E}$ can be inferred from this fact, as explained in Tucker [121] for example.

However, in the context of an implementation in a formal proof system such as Coq which we expect to eventually verify, we would like to minimize the effort in proving each and every specific formula and benefit from the elegance of functional programming to obtain this n-th derivative *recursively*. We would also like for the proof of correctness to only involve facts about derivatives of order 1 when that is possible.

In this subsection, we explore two approaches meeting these criteria. The first one is presented mostly as an oddity which takes advantage of the abstractions of functional programming; however, it is extremely inefficient to implement in practice. The second one is our final implementation; although it has not been proved correct yet, it is both functionally elegant and amenable

to formal proof. Expressions in $\mathcal{E}$ are left implicit in this part to keep formulas legible, but the functions mentioned should still be understood as elements of $\mathcal{E}$ univariate in the same variable x.

A peculiar kind of recursion Let us first describe a surprisingly simple way of doing AD of order n.

Looking back at the previous subsection, what we did was overload common mathematical operators. For example,

$$\begin{aligned} \cdot : \mathbb{R} \times \mathbb{R} &\to \mathbb{R} \\ (x, y) &\mapsto x \cdot y \end{aligned}$$

became

$$\begin{aligned} \cdot_1 : \mathbb{R}^2 \times \mathbb{R}^2 &\to \mathbb{R}^2 \\ ((x_1, x_2), (y_1, y_2)) &\mapsto (x_1 \cdot y_1, x_1 \cdot y_2 + x_2 \cdot y_1). \end{aligned}$$

Let us rename multiplication $\cdot$ as $\cdot_0$. We had the nice property that for all $x \in \mathbb{R}$, for all f and g differentiable at x,

$$(f(x), f'(x)) \cdot_1 (g(x), g'(x)) = (f(x) \cdot_0 g(x), (f(x) \cdot_0 g(x))')$$

We can operate this renaming for all operators described in the previous subsection, for instance $(x, y) +_1 (z, t) = (x + z, y + t)$. Now if we pose

$$\begin{aligned} &\cdot_2 : (\mathbb{R}^2)^2 \times (\mathbb{R}^2)^2 \to (\mathbb{R}^2)^2 \\ &((x_1, x_2), (y_1, y_2)) \mapsto (x_1 \cdot_1 y_1, x_1 \cdot_1 y_2 +_1 x_2 \cdot_1 y_1) \end{aligned}$$

we can see the following: Let $x \in \mathbb{R}$, $f, g : \mathbb{R} \to \mathbb{R}$ twice differentiable at x. For all functions h, let us denote h' for $h(x)$, h'' for $h''(x)$. We have:

$$\begin{aligned} &((f, f'), (f', f'')) \cdot_2 ((g, g'), (g', g'')) = \\ &((f, f') \cdot_1 (g, g'), (f, f') \cdot_1 (g', g'') +_1 (f', f'') \cdot_1 (g, g')) = \\ &((f \cdot_0 g, (f \cdot_0 g)'), ((f \cdot_0 g)', f \cdot_0 g'' + f' \cdot_0 g' + f' \cdot_0 g' + f'' \cdot_0 g')) = \\ &((f \cdot_0 g, (f \cdot_0 g)'), ((f \cdot_0 g)', (f \cdot_0 g)'')) \end{aligned}$$

so that although we have a redundancy in the middle, we now have the derivatives of order 0, 1 and 2 of $f \cdot g$ at x.

In the same way, we could recursively define

$$\begin{aligned} \cdot_{n+1} : (\mathbb{R}^{2^n})^2 \times (\mathbb{R}^{2^n})^2 &\to (\mathbb{R}^{2^n})^2 \\ ((x_1, x_2), (y_1, y_2)) &\mapsto (x_1 \cdot_n y_1, x_1 \cdot_n y_2 +_n x_2 \cdot_n y_1) \end{aligned}$$

as well as for all the other operators of $\mathcal{E}$. An induction on the index n of the operators would show that it is possible by iterating this construct to obtain a vector of exponential size in n containing all of $f(x), f'(x), \cdots, f^{(n)}(x)$. We have iterated the formal construction of automatic differentiation from base operations by applying it recursively to itself. Although this is a pretty result, it is not usable in practice, which is why we move on to a more reasonable way of implementing AD or order n.

Building AD of order n more realistically From now on the notation op_n introduced in subsection 13.1.1 is superseded by the following. Let us define

$$\cdot_n : \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \to \mathbb{R}^{n+1}$$

$$\bar{x}, \bar{y} \mapsto (x_0 \cdot y_0, \cdots, \sum_{i=0}^{k} \binom{k}{i} x_i x_{k-i}, \cdots, \sum_{i=0}^{n} \binom{n}{i} x_i x_{n-i})$$

so that if f, g are two n-times differentiable functions at $x \in \mathbb{R}$, we have

$$(f(x), f'(x), \cdots, f^{(n)}(x)) \cdot_n (g(x), g'(x), \cdots, g^{(n)}(x)) = ((f \cdot g)(x), (f \cdot g)'(x), \cdots, (f \cdot g)^{(n)}(x)).$$

In the same way for $op \in \{+, -, /\}$ we can define:

$$op_n : \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \to \mathbb{R}^{n+1}$$

so that if f, g are two n-times differentiable functions at $x \in \mathbb{R}$,

$$(f(x), f'(x), \cdots, f^{(n)}(x)) \, op_n \, (g(x), g'(x), \cdots, g^{(n)}(x)) = ((f \, op \, g)(x), (f \, op \, g)'(x), \cdots, (f \, op \, g)^{(n)}(x)).$$

and similarly for unary operations $op \in \{\exp, \ln, \cos, \sin, \tan, \arctan\}$:

$$op_n : \mathbb{R}^{n+1} \to \mathbb{R}^{n+1}$$

so that f is a n-times differentiable function at $x \in \mathbb{R}$, $op_n(f(x), f'(x), \cdots, f^{(n)}(x)) = ((op\, f)(x), (op\, f)'(x), \cdots, (op\, f)^{(n)}(x)$.

If we want to inductively define each operation as in Section 13.1.1, we will need a formula for the k-th derivative of each operation for $1 \leq k \leq n$. A first idea is to compute these formulas through iterated symbolic derivation: for example, the formula for the 2 first derivatives of multiplication given by this process is:

$$(x_0, x_1, x_2) \cdot_2 (y_0, y_1, y_2) = (x_0 y_0, x_0 y_1 + x_1 y_0, (x_0 y_2 + x_1 y_1) + (x_1 y_1 + x_2 y_0))$$

This example exposes a major problem with using iterated derivation: expressions can blow up exponentially with n and when evaluating the same computations will be done several times. Furthermore, adding the same expression to itself several times will cause a loss of precision which may make the n-th derivative unusable at best, if it is computed in a rigorous way (through interval arithmetic for example), or completely incorrect at worst if it is computed using floating-point numbers.

An interval version of the symbolic derivation method was programmed in Coq using interval arithmetic and, as expected, it quickly becomes uninformative as n grows. The approach we chose uses a "hardcoded formula" for multiplication in order to avoid this explosion, but tries to be more generic for most operations to facilitate (future) formal proofs. Here is our construction.

- $(x_0, \cdots, x_n) +_n (y_0, \cdots, y_n) := (x_0 + y_0, \cdots, x_n + y_n)$

- $(x_0,\cdots,x_n)\cdot_n(y_0,\cdots,y_n) := (x_0\cdot y_0,\cdots,\sum_{i=0}^{k}\binom{k}{i}x_ix_{k-i},\cdots,\sum_{i=0}^{n}\binom{n}{i}x_ix_{n-i})$
- $(x_0,\cdots,x_n)/_n(y_0,\cdots,y_n) := (x_0,\cdots,x_n)\cdot_n((y_0,\cdots,y_n))^{-1_n}$ (the inverse is considered as belonging to the class of usual functions here, and so it is treated below)
- Let f be any of the other (unary) operators in $\mathcal{E}$. It can be checked that the first derivative of f is also an expression $d_f \in \mathcal{E}$. Let us define f_n recursively:
  - $f_0(x) = f(x)$;
  - $f_{n+1}(x_0,\cdots,x_{n+1}) = f(x_0) :: ((x_1,\cdots,x_{n+1})\cdot_n df_n(x_0,\cdots,x_n))$ where :: denotes concatenation.

We need to justify the correctness of this construction. The first two bullet points are straightforward. The third bullet point only replaces a division by a product with the inverse. For the fourth bullet point, suppose that for some g and x,

$$(x_0,\cdots,x_n) = (g(x),g'(x),\cdots,g^{(n)}(x))$$

$$df_n\,(x_0,\cdots,x_n) = (df(x_0),d'_f(x_0),\cdots,d_f^{(n)}(x_0))$$

Then since $d_f = f'$, we have

$$df_n\,(x_0,\cdots,x_n) = (f'(x_0),f''(x_0),\cdots,f^{(n+1)}(x_0))$$

hence

$$\begin{aligned}&((x_1,\cdots,x_{n+1})\cdot_n df_n(x_0,\cdots,x_n)) =\\&(g'(x),\cdots,g^{(n)}(x))\cdot_n(f'(g(x)),f''(g(x)),\cdots,f^{(n+1)}(g(x)))\end{aligned}$$

so that

$$\begin{aligned}&f(x_0) :: ((x_1,\cdots,x_{n+1})\cdot_n df_n(x_0,\cdots,x_n)) =\\&(f(g(x)),g'(x)\cdot f'(g(x)),\cdots,(x\mapsto g'(x)\cdot f'(g(x)))^{(n)}(x)) =\\&(f(g(x)),(f\circ g)'(x),\cdots(f\circ g)^{(n+1)}(x))\end{aligned}$$

which is exactly what we wanted.

Remark 4. *This technique generalizes to the case when we have a differential equation on f of the shape $f^{(k+1)}(t) = g(t,f(t),f'(t),\cdots,f^{(k)}(t))$, where g is defined as an expression of $\mathcal{E}$, and some initial conditions at x: $y_0 := f(x),\cdots$, $y_k := f^{(k)}(x)$.*

We have implemented an interval version of this method in Coq, on top of CoqInterval. It has not been proved correct yet, however it was used to perform benchmarks on Simpson's method for quadrature.

13.1.2 Test of Simpson's Method in Coq

In Section 12.3, we compared our implementation against some integration software which uses quadrature methods. This was fruitful for correctness and accuracy, but the speed of interpreted Coq code is still much slower than a well-tuned `C` implementation. It makes more sense to compare speeds with a Coq implementation of a quadrature method.

Our implementation uses the same strategy of dichotomy as the one described in Section 12.1.4, but uses our implementation of Simpson's method instead of Taylor Models. There are two variants: one which uses the simple version of Simpson's method as in Theorem 7, and one which uses the compound rules with n points as in Theorem 9.

First, we looked at the group of the three first integrals from Section 12.3. They have in common that the integrand is four times differentiable on the whole integration interval. For each of them, we started with the parameters from the corresponding array in Section 12.3 for the line with the best accuracy. The simple Simpson rule being always slower than Taylor Models in practice, we then tried to vary the depth and number of points to reach the fastest possible execution reaching comparable accuracy as the Taylor Model based one. We indicate the target accuracy, depth of the dichotomy, and precision of intermediary computations used to obtain the new result. The column "Points" gives the number of points used in Simpson's compound rule. Results are comparable with an edge in favor of Simpson's method.

Integral	Time	Time (TM)	Error	Depth	Prec.	Points
$\int_0^1 \frac{dx}{1+x^2}$	1.9	2.9	10^{-18}	6	70	55
$\int_0^1 \frac{\arctan\sqrt{x^2+2}}{\sqrt{x^2+2}\,(x^2+1)}dx$	11.854	11.2	10^{-15}	6	55	25
$\int_0^\pi \frac{x\sin x}{1+\cos^2 x}dx$	20.2	27.2	10^{-15}	6	65	25

The second group of tests was more problematic as the integrands are not differentiable on the whole domain. Just like in the case of Rigorous Polynomial Approximations, when this was the case on a subinterval, we reverted to the naive method. For lack of space in the table below, we redefine here

$$I_1 := \int_0^1 \left|\left(x^4 + 10x^3 + 19x^2 - 6x - 6\right) e^x\right| dx$$

and

$$I_2 := \int_{-1}^1 \left(2048x^{12} - 6144x^{10} + 6912x^8 - 3584x^6 + 840x^4 - 72x^2 + 1\right) \exp\left(-\left(x - \tfrac{3}{4}\right)^2\right) \sqrt{1 - x^2}\, dx.$$

In both cases, the input depth has to be severely constrained compared to Section 12.3 in order to get a result in a reasonable time. The column "Accy" corresponds to the accuracy in bits which was asked for and is the same as with Taylor Models.

For I_1, the computation took 33 seconds instead of 2.7, using the same (asked) accuracy of 2^{-54}, using 50 points in the Simpson compound rule but only allowing a depth of 10 instead of 28 in the dichotomy.

For I_2, we reduced the depth from 35 to 10, only computing Simpson with 10 points and still the computation takes 204 seconds, and returns a result with an accuracy of 10^{-4} instead of 10^{-15}.

Int.	Time	Time (TM)	Error	Actual Accy	Depth	Prec.	Points
I_1	33	2.7	10^{-15}	10^{-4}	10	70	50
I_2	204	112	10^{-15}	10^{-4}	10	70	10

We can only conclude that in both cases, Rigorous Polynomial Approximations give more accurate results earlier in the binary search; thanks to this property, giving a high depth parameter like 28 or 35 does not penalize the search since this is only used around the singularities. However, Simpson's method doesn't seem so effective in reducing the breadth of search and we are forced to constrain the depth more, which in turns gives much less accurate results.

These results should be taken with caution as they are computed using an unproved implementation which could very well have flaws and inefficiencies. However, it seems to indicate that a quadrature method such as Simpson's method has the same order of magnitude as our methods based on Rigorous Polynomial Approximations. Our observations seem to be supported by Rall and Corliss [34] whose FORTRAN implementation in the 1980s of a similar scheme was competitive with routines from the then state-of-the-art QUADPACK numerical integration library also written in FORTRAN.

Chapter 14

Conclusion and Perspectives

In this part, we have presented a method for computing and formally verifying numerical enclosures of univariate definite integrals inside the Coq proof assistant. This method deals with both proper and improper integrals. It has been integrated into the `interval` tactic of CoqInterval. In the same computation step, it proves that integrals exist and provides formally verified enclosures of their value. These proofs rely on the formal theory of generalized Riemann integrals provided by our extension to the Coquelicot library. In the proper case, the enclosing method only requires that there exist a Rigorous Polynomial Approximation of the integrand, so that its only limit is the underlying CoqInterval library. Any new functions added to the set of expressions $\mathcal{E}$ handled by the library would be almost immediately supported.

The current treatment of improper integrals is less automated. In particular, the syntactic expression of the integrand has to make explicit the element of the appropriate scale which models its asymptotic behavior near the singularity. The tactic currently supports two scales, the exponential $e^{\gamma x}$ and $x^\alpha \ln^\beta x$. We could provide more scales to users, or at least merge these two into the more common $e^{\gamma x} x^\alpha \ln^\beta x$ scale which has the advantage of being totally ordered with respect to the "small o" relation $o(\cdot)$ at e.g. $+\infty$.

One possible direction to automate the discovery of asymptotics could then be to build yet another interpretation for elements in $\mathcal{E}$ in terms of their asymptotic behavior. Suppose for example that $u > 0$ and we define a function $[\![\cdot]\!]_{[u;+\infty)} : \mathcal{E} \to \mathbb{I} \times \mathbb{Z}^3$ with the specification that for all real expressions $e(x) \in \mathcal{E}$, if $[\![e]\!]_{[u;+\infty)} = (\mathbf{x}, (\alpha, \beta, \gamma))$ then there exists a continuous function f such that

$$\forall x \in [u; +\infty), f(x) \in \mathbf{x} \text{ and } [\![e(x)]\!]_{\mathbb{R}} = f(x) \cdot x^\alpha \ln^\beta(x)\, e^{\gamma x}. \tag{14.1}$$

We would need to explain how this interpretation propagates along the operators of $\mathcal{E}$. For example, multiplication is straightforward: if $[\![e_1]\!]_{[u;+\infty)} = (\mathbf{x_1}, (\alpha_1, \beta_1, \gamma_1))$ and $[\![e_2]\!]_{[u;+\infty)} = (\mathbf{x_2}, (\alpha_2, \beta_2, \gamma_2))$, then $[\![e_1 \cdot e_2]\!]_{[u;+\infty)}$ could be defined as $(\mathbf{x_1} \cdot \mathbf{x_2}, (\alpha_1 + \alpha_2, \beta_1 + \beta_2, \gamma_1 + \gamma_2))$.

A more difficult example is addition, with two cases. In the easy case, if $(\alpha_1, \beta_1, \gamma_1) = (\alpha_2, \beta_2, \gamma_2)$ then $[\![e_1 + e_2]\!]_{[u;+\infty)} = (\mathbf{x_1} + \mathbf{x_2}, (\alpha_1, \beta_1, \gamma_1))$ works. However, if $x^{\alpha_1} \ln^{\beta_1}(x)\, e^{\gamma_1 x} = o(x^{\alpha_2} \ln^{\beta_2}(x)\, e^{\gamma_2 x})$ at $+\infty$, then if f_1 and f_2 are

the continuous functions from the specification (14.1) for e_1 and e_2 respectively, for $x \in [u;\infty)]$:

$$\begin{aligned} e_1(x) + e_2(x) =& f_1(x) \cdot x^{\alpha_1} \ln^{\beta_1}(x)\, e^{\gamma_1 x} + f_2(x) \cdot x_2^{\alpha} \ln^{\beta_2}(x)\, e^{\gamma_2 x} \\ &= \Big(f_1(x) \cdot x^{\alpha_1 - \alpha_2} \ln^{\beta_1 - \beta_2}(x)\, e^{(\gamma_1 - \gamma_2)x} + f_2(x) \Big) \cdot x^{\alpha_2} \ln^{\beta_2}(x)\, e^{\gamma_2 x}. \end{aligned}$$

Since $x^{\alpha_1-\alpha_2} \ln^{\beta_1-\beta_2}(x)\, e^{(\gamma_1-\gamma_2)x}$ is continuous and bounded, we would then be able to compute a new interval $\mathbf{y}$ such that

$$[\![e_1(x) + e_2(x)]\!]_{[u;+\infty)} = (\mathbf{y}, (\alpha_2, \beta_2, \gamma_2))$$

while being correct with respect to the specification in Formula (14.1). This is possible because the scale constituted by the $x^{\alpha} \ln^{\beta}(x)\, e^{\gamma x}$ is totally ordered with respect with the relation $o(\cdot)$.

After we would program a procedure which computes $\int_u^{\infty} x^{\alpha} \ln^{\beta}(x)\, e^{\gamma x}\, dx$ given α, β and γ, we would then treat previously inaccessible examples at the current stage of the code as seen in Section 12.2.4.

In the introduction of this part, we opposed *specification* to *validation* of computations. However, there need not be such a clear cut between the two notions, and parts of our computations in the context of integrals could be delegated to an external oracle. For example, the "depth tree" describing where to use dichotomy in an interval when computing a proper integral could be provided by a fast Ocaml program. Similarly, the initial estimate used to find the absolute error from the relative error provided by the user in the `integral` tactic need not be proved and we can replace it by the approximation part (without the error) of Simpson's method.

Before writing the programs and tactics described in Section 12, we wrote prototypes in Ocaml, as well as other routines for solving Ordinary Differential Equations using Taylor Models. It might seem at first that in the absence of a need for complex built-in features, prototyping in Ocaml before writing a Coq program would be a duplication of effort. However, certain realities of Coq development made this orders of magnitude faster in our case.

First, Coq is not such a great testing environment: computations are slow, there is no profiling, logging or printing possible[1] and in particular no pretty-printing, for example of floats. These aspects of programming are crucial for rapid development.

Secondly, since the libraries we could have used for prototyping are geared towards their use in tactics, most of what they do is handled by Ltac which is untyped, unhelpful when it fails, and needs a lot of side-knowledge to be made to work. Alternatives like Mtac [133] may be steps in the right direction.

Nested integrals are not supported by our method. The naive enclosure approach could easily be adapted to support them, but performances would be even worse due to the curse of dimensionality. As there exists no general approach for integrating multivariate polynomials,[2] being able to compute rigorous multivariate polynomial approximations would presumably not help.

[1] although this is changing with a recent pull request `https://github.com/coq/coq/pull/714`

[2] Any 3-SAT instance can be reduced to approximating the integral of a multivariate polynomial.

While our adaptive bisection algorithm and our rigorous quadrature based on primitives of polynomials might seem crude, they proved effective in practice. They produce accurate approximations of non-pathological integrals in a few seconds, and thus they are usable in an interactive setting. Moreover, they are able to handle functions with unbounded second derivatives in a rigorous way, as well as unbounded integration domains. Another contribution is the way we are able to infer that a function is integrable from a successful computation of its integral.

Following Rall & Corliss [34], we could have extended this adaptivity even more to include the *order* chosen for each subinterval.

We could also have tried a much more general method, that is, solving a differential equation built from the integrand, as we did with VNODE-LP. There has been some work done for Coq in the setting of exact real arithmetic [89] [99], but the performances are not good enough in practice. Much closer to actual numerical methods is Immler's work in Isabelle/HOL [76], which uses an arithmetic on affine forms. This approach is akin to computing with degree-1 RPAs. Note that such computations in Isabelle are treated in a slightly different way from Coq: they are performed by an extracted Ocaml program whose result is then imported inside the proof assistant.

Part IV

Conclusion

In this concluding part, we discuss the general question of when and how to trust proofs relying on computations. Then, we suggest how the domains of formal methods and cryptocurrencies might benefit each other.

Can we trust computations used in mathematical proofs?

Despite numerical integration being an old problem and despite the relative simplicity of the mathematical ingredients of our methods, we easily found mistakes on elementary examples. One cannot exclude other mistakes in numerical computations in published works. We can speculate that most of the time, as in the case of the slightly off integral we found in the Ternary Goldbach Conjecture, they are only slightly wrong and do not impact the correctness of results. However, not only is there no guarantee that this is always the case, but this level of uncertainty is not acceptable.

In both cases, the mistakes we found happened despite using state-of-the-art integration software. However, computation-based proofs which use custom-written code are not a rare occurrence. In that case, there are even further reasons to be circumspect. We closely examined the proof of correctness of the `C++` code in the original proof of the Double Bubble conjecture [66], and it turns out that the pseudo-code described in the paper has discrepancies with the distributed code. For example in Figure 14.1, the value assigned to the variable R is $\frac{\frac{\sqrt{3}}{2}}{-H_i - \frac{1}{Y}}$ in the `C++` code and $\frac{1}{-H_i - \frac{1}{Y}}$ in the pseudocode.

Figure 14.1: Discrepancy between the pseudo-code in the paper and the distributed C++ code for the Double Bubble Conjecture

```
        // if the endpoint is too fat, go back and subdivide input
        if (C_2.width() > .5) return NORESULT;

// step 6
        // set Global.Y_min,Global.Y_max for use by integration routines
        Global.Y_min = - F_i / (1 + Sqrt(1 + F_i*H_i)); // local y min
        Global.Y_max = (1 + Sqrt(1 + F_o*H_o)) / H_o;          // local y max
        rp W_ends = ((1-C_1)^2)*(2+C_1)/3 + ((1-C_2)^2)*(2+C_2)/3;
        rp Y = Compare(C_1,Root3/2,Global.Y_min,Y_1);
        if (Y * H_i < -1 && Y > 0)
        {
                // test for inefficient torus
                rp R = (Root3 /█2) / (-H_i - 1/Y);
                rp W = 2.5 * (R^2) * (Y + R*Root3/2);   // upper bd for vol/pi
                if (W < W_ends) return REJECT(6);
        }

// step 7
        // calc some integration endpoints
        rp Y_left = Sqrt(F_o/H_o);
        if (Global.Y_min < Y_2 && Y_left < Global.Y_max)
                Y_left = Max(Y_1,Y_left);
```

```
if Width(C_2) > .5 then return NORESULT

W_ends := (1 - C_1)^2(2 + C_1)/3 + (1 - C_2)^2(2 + C_2)/3
Y_min := -F_i/(1+ Sqrt(1 + F_i H_i))
Y_max := (1+Sqrt(1 + F_o H_o))/H_o
Y := Compare(C_1,Sqrt(3)/2, Y_min, Y_1)
if Y H_i < -1 then
     begin
     R := 1/(-H_i - 1/Y)
     W := 2.5(Y + (Sqrt(3)/2)R)R^2
     if W < W_ends then return REJECT
     end

Y_left := Sqrt(F_o/H_o)
if (Y_min < Y_2 and Y_left < Y_max) then Y_left := M
     else return NORESULT
```

We re-wrote the pseudo-code in Ocaml using our own interval arithmetic library and barring an error on our side, it is not semantically equivalent. However, when we changed the parameters scrupulously to the actual `C++` code we were able to replicate their results. Thus, it is difficult to assess whether these were harmless changes with a justification that did not make it to the paper or if there were typos in the code, rendering the proof invalid. This seems to validate *a posteriori* the decision of the reviewers of the first proof of the Kepler conjecture which relied on 3 Gigabytes of code [60] to assert that they could not certify the correctness of the proof: The Double Bubble code is only 849 lines or 21 kilobytes of code.

We acknowledge that there is no satisfactory tool, as of now, which will meet the computational needs of *all* modern mathematical proofs at the level of assurance of formal proofs. There are already a range of certified tools which tackle specific domains, such as solving differential equations numerically [76], proving nonlinear inequalities [84], checking satisfiability [41], or checking primality [58]. It is possible that some types of computations will remain out of reach of proof

assistants in the near future. For these, one may either develop a new tool to produce formally verified results or fall back on unverified computations.

We contributed to this Swiss-knife-like array of tools destined to the verification of proofs relying on demanding computations, by providing a methodology to validate the recurrences outputted by creative telescoping algorithms. However, in this case, this validation is not as automatic as we had hoped it to be. This work is an example of another problem: the program may be doing exactly what it is supposed to do, but do the semantics of the object it produces match with those of the object we think we are reasoning on? For this reason, using a proof assistant is of precious help.

Some Remarks Concerning Tools for Formal Proofs

In Parts II and III, we commented on the ease or difficulty of using a proof assistant. If it may seem at times that our comments are too negative, this is also because of our enthusiasm and eagerness for improvements to theorem proving. One should also keep in mind that our experience was mainly with the Coq proof assistant and in particular with the Mathematical Components libraries, using the SSReflect proof language; surely other proof assistants provide a better experience on some aspects, and worse on others.

During an internship in the summer of 2016, we used the Lean theorem prover [37] to attempt to port some of Mathematical Components's elementary group theory. Lean is based on the Calculus of Inductive Constructions, like Coq, but it is comparatively very young. For this reason, at the time of this internship, the tactic language was still rudimentary and few libraries were available: this made the task very hard, and we did not succeed in proving Sylow's theorem as we had intended in the beginning. This is of course no fault of Lean's, which has been evolving at a remarkable speed, but it is a testimony to how much combined work has been put into making theorem proving in Coq both effective and agreeable.

Formal Proofs and Cryptocurrencies

We would like to discuss how the domain of computer-aided proofs, with its tools and principles can benefit the emerging domain of blockchains and cryptocurrencies, but also how this domain raises new challenges.

In the domain of formal proofs, as ruthless as the peer review process may sometimes be, authors and reviewers are in a relationship of trust. We mean this in the sense that even though the reviewer carefully checks all of the author's claims, she does not assume that the person who wrote the proof is actively trying to trick them into thinking that they have proved a certain theorem, when in fact they have either proved a different theorem or made use of a bug to build a fake proof.

In 2014, on a website called Proof Market which set bounties in the digital currency Bitcoin for Coq proofs of theorems submitted by users, someone managed to walk away with 1 Bitcoin (about 600 euros at the time) by providing a proof of `False` along the lines of the following script:

```
Definition False := True.

Lemma foo : False.
```

```
Proof.
unfold foo; trivial.
Qed.
```

This website created a completely new setting for formal proofs, where an automated checker would trigger a transaction upon receiving (what it thought to be) a valid proof. Since no human was in the loop, this created an incentive to find bugs in the checker rather than to write a correct proof.

A Blockchain is a linked list of blocks such that the $i+1$-th block contains a cryptographic hash of the i-th block. Blockchains are typically built using a distributed protocol which gives the right to add a new block to a node selected randomly in proportion of their commitment in the form of some rare resource (such as computing power, or money, or storage, etc...). This scheme was invented by Satoshi Nakamoto who created the first decentralized digital currency, Bitcoin, where blocks contain lists of transactions [97]. Several improvements have been proposed since, including the idea of a *general purpose* cryptocurrency where transactions contain function calls and blocks additionally maintain the state of a global decentralized computer [130]. In particular, the notion of "smart contracts", which are autonomous program-agents imagined in 1995 by Nick Szabo, became a reality in the shape of autonomous pieces of code "running" on the blockchain. In light of several high-profile hacks and thefts of tens of millions of euros in recent years, providing formal guarantees on the behavior of such code, which can typically manipulate considerable amounts of money or handle authorization of access to other contracts, has become an urgent problem.

We have participated in a Microsoft Research hackathon in which we built a proof-of-concept program capable of analyzing both high-level and low-level code of smart contracts in the Ethereum cryptocurrency [15]. We used the F* [119] functional language.

We also discovered the Easycrypt [9] proof assistant, geared towards proofs on cryptographic protocols and more generally on probabilistic programs. We formalized a model called "the Bitcoin Backbone Protocol" [47], an attempt at proving elementary properties of the Bitcoin protocol, together with Pierre-Yves Strub. However, a complete formal proof is likely to necessitate improvements to Easycrypt.

Finally, we did some work in formalizing in Coq the semantics of the smart-contract language of another general-purpose cryptocurrency called Tezos [53]. The main difference with Ethereum is that Tezos proposes formal semantics of this language, which, like Ocaml, is strongly typed and purely functional.

To sum up, we think this is a new type of context for formal proofs, because of the unique adversarial model. When one checks a formal development, one does not usually assume malice from the person who wrote it. By contrast, if someone anonymous manages to take advantage of a weakness in a theorem prover to pretend that a smart contract has a certain behavior, they could walk away with millions with no chance of ever getting caught. Specific tools to tackle such security challenges are being developed.

Bibliography

[1] Mathematical Components Libraries. http://math-comp.github.io/math-comp/, 2013. Version 1.4. For Coq 8.4pl3.

[2] Jounaidi Abdeljaoued and Henri Lombardi. *Méthodes matricielles - Introduction à la complexité algébrique*. Mathématiques et Applications. Springer, 2003.

[3] Zafar Ahmed. Ahmed's integral: the maiden solution. *Mathematical Spectrum*, 48(1):11–12, 2015.

[4] Kenneth Appel, Wolfgang Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

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

[6] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9(1), December 2007.

[7] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.

[8] Henk Barendregt and Arjeh M Cohen. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *Journal of Symbolic Computation*, 32(1):3–22, 2001.

[9] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, pages 146–166. Springer, 2014.

[10] François Bergeron and Simon Plouffe. Computing the generating function of a series given its first few terms. *Experimental mathematics*, 1(4):307–312, 1992.

[11] Sophie Bernard, Yves Bertot, Laurence Rideau, and Pierre-Yves Strub. Formal proofs of transcendence for e and π as an application of multivariate and symmetric polynomials. *CoRR*, abs/1512.02791, 2015.

[12] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of gmp square root. *Journal of Automated Reasoning*, 29(3):225–252, Sep 2002.

[13] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *TYPES'06*, LNCS, pages 48–62, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In *ACM Workshop on Programming Languages and Analysis for Security*, Vienna, Austria, October 2016.

[16] Jesse Bingham. Formalizing a proof that e is transcendental. *Journal of Formalized Reasoning*, 4(1):71–84, 2011.

[17] Dario Bini, Milvio Capovani, Francesco Romani, and Grazia Lotti. $\mathrm{O}(\mathrm{n}^{2.7799})$ complexity for n*n approximate matrix multiplication. *Inf. Process. Lett.*, 8(5):234–235, 1979.

[18] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *Certified Programs and Proofs*, Certified Programs and Proofs, Kenting, Taiwan, December 2011. Springer.

[19] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal Proof of a Wave Equation Resolution Scheme: the Method Error. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the first Interactive Theorem Proving Conference (ITP)*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, July 2010. Springer. (merge of TPHOL and ACL2).

[20] 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, 2015.

[21] Nicolas Bourbaki. *Eléments de mathématique : Livre III. Topologie générale*. Hermann, 1968.

[22] Peter Bürgisser, Michael Clausen, and Amin Shokrollahi. *Algebraic complexity theory*, volume 315. Springer Science & Business Media, 2013.

[23] Henri Cartan. Filtres et ultrafiltres. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences, Paris*, 205:777–779, 1937.

[24] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[25] Frédéric Chyzak. *The ABC of Creative Telescoping—Algorithms, Bounds, Complexity.* PhD thesis, Ecole Polytechnique X, 2014.

[26] Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi. A Computer-Algebra-Based Formal Proof of the Irrationality of $\zeta(3)$. In *ITP - 5th International Conference on Interactive Theorem Proving*, Vienna, Austria, 2014.

[27] Frédéric Chyzak, Bruno Salvy, and Paul Zimmerman. Algolib. http://algo.inria.fr/libraries/, 2013. Version 17.0. For Maple 17.

[28] Cyril Cohen. Construction of real algebraic numbers in Coq. In *ITP*, volume 7406 of *LNCS*. Springer, August 2012.

[29] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, Ecole Polytechnique X, 2012.

[30] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs*, pages 147 – 162, Melbourne, Australia, December 2013.

[31] Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1:02):1–40, February 2012.

[32] Henri Cohen. Démonstration de l'irrationalité de ζ (3)(d'apres apéry). *Séminaire de Théorie des Nombres, Grenoble*, 1979, 1978.

[33] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987.

[34] George F. Corliss and Louis B. Rall. Adaptive, self-validating numerical quadrature. *SIAM Journal on Scientific and Statistical Computing*, 8(5):831–847, 1987.

[35] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *Mathematical Knowledge Management*, pages 88–103. Springer, 2004.

[36] Philip J. Davis and Philip Rabinowitz. Chapter 4 - error analysis. In Philip J. Davis and Philip Rabinowitz, editors, *Methods of Numerical Integration (Second Edition)*, pages 271 – 343. Academic Press, second edition edition, 1984.

[37] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. *The Lean Theorem Prover (System Description)*, pages 378–388. Springer International Publishing, Cham, 2015.

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

[39] Jean-Guillaume Dumas and Victor Pan. Fast matrix multiplication and symbolic computation. *CoRR*, abs/1612.05766, 2016.

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

[41] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds, and Cesare Tinelli. Extending smtcoq, a certified checker for smt. *arXiv preprint arXiv:1606.05947*, 2016.

[42] Leonhard Euler. Observations on a certain theorem of Fermat and on others concerning prime numbers. *ArXiv Mathematics e-prints*, January 2005.

[43] 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.

[44] 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.

[45] Laurent Fousse. *Intégration numérique avec erreur bornée en précision arbitraire.* PhD thesis, Université Henri Poincaré-Nancy I, 2006.

[46] Johannes Fürlinger and Josef Hofbauer. q-catalan numbers. *Journal of Combinatorial Theory, Series A*, 40(2):248–264, 1985.

[47] Juan A Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, pages 281–310, 2015.

[48] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[49] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. working paper or preprint, March 2009.

[50] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics*, pages 333–333. Springer, 2008.

[51] Georges Gonthier. Point-free, set-free concrete linear algebra. In *International Conference on Interactive Theorem Proving*, pages 103–118. Springer, 2011.

[52] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le 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, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[53] L.M. Goodman (pseudonym). Tezos: A self-amending crypto ledger position paper, 2014.

[54] Michael Gordon, Robin Milner, and CP Wadsworth. Edinburgh LCF: a mechanized logic of computation, volume 78 of. *Lecture Notes in Computer Science*, pages 1–6, 1979.

[55] Xavier Gourdon. The 10^{13} first zeros of the Riemann Zeta function, and zeros computation at very large height, 2004. *Unpublished, Available: http://numbers. computation. free. fr/Constants/constants. html (accessed 22 January 2012)*, 2004.

[56] Timothy Gowers and Michael Nielsen. Massively collaborative mathematics. *Nature*, 461(7266):879–881, 2009.

[57] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 235–246, New York, NY, USA, 2002. ACM.

[58] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. *A Computational Approach to Pocklington Certificates in Type Theory*, pages 97–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[59] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Hang Nguyen, et al. A formal proof of the Kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

[60] Thomas C Hales. The Kepler conjecture. *arXiv preprint math.MG/9811078*, 1998.

[61] Thomas C Hales. Mathematics in the age of the Turing machine., 2014.

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

[63] John Harrison. Formalizing an analytic proof of the Prime Number Theorem. *Journal of Automated Reasoning*, 43:243–261, 2009.

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

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

[66] Joel Hass and Roger Schlafly. Double bubbles minimize. *Annals of Mathematics. Second Series*, 151(2):459–515, 2000.

[67] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, July 1998.

[68] Dennis A. Hejhal and Andrew M. Odlyzko. Alan Turing and the Riemann zeta function. *Alan Turing: His Work and Impact*, pages 265–279, 2012.

[69] Harald A. Helfgott. Rigorous numerical integration. MathOverflow. URL:https://mathoverflow.net/q/123677 (version: 2013-03-05).

[70] Harald A. Helfgott. Major arcs for Goldbach's problem. http://arxiv.org/abs/1305.2897, 2014.

[71] Harold A. Helfgott. The ternary Goldbach conjecture is true (2013). *arXiv preprint arXiv:1312.7748*, 2014.

[72] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. *CoRR*, abs/1605.00723, 2016.

[73] Florent Hivert. Coq-combi. https://github.com/hivert/Coq-Combi.

[74] Johannes Hölzl, Fabian Immler, and Brian Huffman. *Type Classes and Filters for Mathematical Analysis in Isabelle/HOL*, pages 279–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[75] Michael Hutchings, Frank Morgan, Manuel Ritoré, and Antonio Ros. Proof of the double bubble conjecture. *Annals of Mathematics*, pages 459–489, 2002.

[76] Fabian Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *NASA Formal Methods (NFM)*, volume 8430 of *LNCS*, pages 113–127. Springer, 2014.

[77] Mioara M. Joldes. *Rigorous Polynomial Approximations and Applications*. Theses, Ecole normale supérieure de lyon - ENS LYON, September 2011.

[78] Mikhail. M. Kapranov and Vladimir A. Voevodsky. ∞-groupoids and homotopy types. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 32(1):29–46, 1991.

[79] Donald E. Knuth. Dancing links. *arXiv preprint cs/0011047*, 2000.

[80] Leslie Lamport. *LATEX: a document preparation system: user's guide and reference manual*. Addison-wesley, 1994.

[81] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014.

[82] Catherine Lelay. *Repenser la bibliothèque réelle de Coq: vers une formalisation de l'analyse classique mieux adaptée*. PhD thesis, Université Paris Sud-Paris XI, 2015.

[83] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 3.12): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2011.

[84] Victor Magron. Nlcertify: A tool for formal nonlinear optimization. In *International Congress on Mathematical Software*, pages 315–320. Springer, 2014.

[85] Assia Mahboubi and Benjamin Gregoire. Proving equalities in a commutative ring done right in Coq. In *TPHOLs 2005*, volume 3603 of *LNCS*, pages 98–113, Oxford, United Kingdom, August 2005. Springer.

[86] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. In Jasmin Christian Blanchette and Stephan Merz, editors, *Proceedings of the 7th Conference on Interactive Theorem Proving*, volume 9807 of *Lecture Notes in Computer Science*, pages 274–289, Nancy, France, 2016.

[87] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *Accepted in Journal of Automated Reasoning, S.I : ITP 2016*, 2017.

[88] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer.

[89] Evgeny Makarov and Bas Spitters. The Picard algorithm for ordinary differential equations in Coq. In *Interactive Theorem Proving (ITP)*, LNCS, pages 463–468. Springer, 2013.

[90] Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *Int. J. Pure Appl. Math*, 2003.

[91] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, pages 1–31, 2015.

[92] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, December 2001.

[93] Micaela Mayero. Using theorem proving for numerical analysis. *Lecture notes in computer science*, pages 246–262, 2002.

[94] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[95] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron, and Paul DeMarco. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada, 2005.

[96] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, Philadelphia, PA, USA, 2009.

[97] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[98] Nedialko S. Nedialkov. Interval tools for ODEs and DAEs. In *Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN)*, 2006. http://www.cas.mcmaster.ca/~nedialk/vnodelp/.

[99] Russell O'Connor and Bas Spitters. A computer verified, monadic, functional implementation of the integral. *Theoretical Computer Science*, 411(37):3386–3402, 2010.

[100] Victor Pan. *How to multiply matrices faster*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[101] Lawrence C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[102] Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger. $A = B$. A K Peters Ltd., Wellesley, MA, 1996.

[103] J.D. Phillips and David Stanovskỳ. Using automated theorem provers in nonassociative algebra, 2008.

[104] Titus Piezas. Ramanujan's constant (e $\pi \sqrt{163}$) and its cousins.

[105] D.H.J. Polymath. The "bounded gaps between primes" polymath project-a retrospective. *arXiv preprint arXiv:1409.8361*, 2014.

[106] Loïc Pottier. Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. *CoRR*, abs/1007.3615, 2010.

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

[108] Werner Romberg. Vereinfachte numerische integration. *Det Kongelige Norske Videnskabers Selskab Forhandlinger*, 28(7):30–36, 1955.

[109] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010. http://www.ti3.tu-harburg.de/rump/intlab/.

[110] Rémy El Sibaïe and Jean-Christophe Filliâtre. Combine: an ocaml library for combinatorics, 2016.

[111] 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.

[112] Bruno Salvy and Paul Zimmermann. Gfun: a maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software (TOMS)*, 20(2):163–177, 1994.

[113] Arnold Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[114] Carlos Simpson. Homotopy types of strict 3-groupoids. *arXiv:math/9810059*, October 1998.

[115] Joel Spolsky and Jeff Atwood. Math Overflow. https://mathoverflow.net/, 2008.

[116] Andrew James Stothers. *On the complexity of matrix multiplication*. PhD thesis, The University of Edinburgh, 2010.

[117] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[118] Volker Strassen. Relative bilinear complexity and matrix multiplication. *Journal für die reine und angewandte Mathematik*, 375:406–443, 1987.

[119] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[120] Warwick Tucker. The Lorenz attractor exists. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 328(12):1197–1202, 1999.

[121] Warwick Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press, Princeton, NJ, USA, 2011.

[122] Simon Henry (https://mathoverflow.net/u/22131/). What is the mistake in the proof of the homotopy hypothesis by Kapranov and Voevodsky? MathOverflow. Url: https://mathoverflow.net/q/234492 (version: 2017-11-29).

[123] 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.

[124] Vladimir A. Voevodsky. Univalent foundations. http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf.

[125] Alfred North Whitehead. *An introduction to mathematics*. Courier Dover Publications, 2017.

[126] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[127] Virginia V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898. ACM, 2012.

[128] Shmuel Winograd. On multiplication of $2\times$ 2 matrices. *Linear algebra and its applications*, 4(4):381–388, 1971.

[129] Wen-Jin Woan, Lou Shapiro, and D.G. Rogers. The catalan numbers, the lebesgue integral, and 4n-2. *American Mathematical Monthly*, pages 926–931, 1997.

[130] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2013.

[131] Doron Zeilberger. A holonomic systems approach to special functions identities. *J. Comput. Appl. Math.*, 32(3):321–368, 1990.

[132] Doron Zeilberger. The method of creative telescoping. *J. Symbolic Comput.*, 11(3):195–204, 1991.

[133] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. *SIGPLAN Not.*, 48(9):87–100, September 2013.

[134] Vadim 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.

Appendix A

Complexity of Some Matrix Multiplication Algorithms

The following tables were deferred to the Appendix from Chapter 4. Table A.2 concerns the following tensor from Pan, which we present as a table:

```
Table T13_1 :

Row t13_1_row1 :
Seq(
[
a[i,0];
b[0,k]*epsilon^3;
c[i,k]*epsilon^5
],

i=0..m-1,
k=0..p-1
)

Row t13_1_row2 : Seq(
[
u[0,k]*epsilon^4;
v[k,i]*epsilon^4;
w[0,i]
],
i=0..m-1,
k=0..p-1
)

Row t13_1_row3 : Seq(
[
x[k,i]*epsilon^6;
y[i,0];
z[k,0]*epsilon^2
],
i=0..m-1,
k=0..p-1
)

Table T13_2 :

Row t13_2_row1 : Seq(
```

```
[
-a[i,0];
-bb[0,k]*epsilon^3;
cc[i,k]*epsilon^5
],
i=0..m-1,
k=0..p-1
)

Row t13_2_row2 : Seq(
[
u[1,k]*epsilon^4;
v[k,i]*epsilon^4;
w[1,i]
],
i=0..m-1,
k=0..p-1
)

Row t13_2_row3 : Seq(
[
xx[k,i]*epsilon^6;
yy[i,0];
z[k,0]*epsilon^2
],
i=0..m-1,
k=0..p-1
)

Table T13_3 :
Row t13_3_row1 : Seq(
[
-a[i,0];
(sum((1/p)*b[0,eb],eb=1..p-1))*epsilon^3;
0
],
[
i=0..m-1
]
)



Row t13_3_row2 : Seq(
[
0;
(sum((1/p)*v[ev1,i],ev1=1..p-1))*epsilon^4;
p*w[0,i]
],
[
i=0..m-1
]
)


Row t13_3_row3 : Seq(
[
0;
y[i,0];
sum(z[ez,0],ez=1..p-1)*epsilon^2
],
```

```
[
i=0..m-1
]
)

Table T13_4 :

Row t13_4_row1 : Seq(
[
a[i,0];
(sum((- (1/p))*bb[0,ebb],ebb=1..p-1))*epsilon^3;
0
],
[
i=0..m-1
]
)

Row t13_4_row2 : Seq(
[
0;
(sum((1/p)*v[ev2,i],ev2=1..p-1))*epsilon^4;
p*w[1,i]
],
[
i=0..m-1
]
)

Row t13_4_row3 : Seq(
[
0;
yy[i,0];
sum(z[ez,0],ez=1..p-1)*epsilon^2
],
[
i=0..m-1
]
)

;;
Constraints :
a[i:=0,0] = - sum(a[ha,0],ha=1..m-1) (* constr_a_1 *)
w[0,i:=0] = - sum(w[0,hw0],hw0=1..m-1) (* constr_w_1 *)
w[1,i:=0] = - sum(w[1,hw1],hw1=1..m-1) (* constr_w_2 *)
y[i:=0,0] = - sum(y[hy,0],hy=1..m-1) (* constr_y_1 *)
yy[i:=0,0] = - sum(yy[hyy,0],hyy=1..m-1) (* constr_y_2 *)

u[0,k:=0] = - sum(u[0,eu0],eu0=1..p-1) (* constr_u_1 *)
u[1,k:=0] = - sum(u[1,eu1],eu1=1..p-1) (* constr_u_2 *)

c[i,k:=0] = - sum(c[i,ec],ec=1..p-1)   (* constr_c_1 *)
cc[i,k:=0] = - sum(cc[i,ecc],ecc=1..p-1) (* constr_c_2 *)
x[k:=0,i] = - sum(x[ex,i], ex=1..p-1)   (* constr_x_1 *)
xx[k:=0,i] = - sum(xx[exx,i],exx= 1..p-1) (* constr_x_2 *)

c[i:=0,k] = 0
cc[i:=0,k] = 0
v[k,i:=0] = 0
x[k,i:=0] = 0
xx[k,i:=0] = 0
b[0,k:=0] = 0
```

```
bb[0,k:=0] = 0
z[k:=0,0] = 0
v[k:=0,i] = 0



(* substitutions *)
a[i,k:=0] = a1[i-1,0] (* instSubsa *)
b[i:=0,k] = b1[0,k-1] (* instSubsb *)
bb[i:=0,k] = b1[0,k+p-2] (* k-1 + (p-1) instSubsbb*)
c[i,k] = c1[i-1,k-1] (* instSubsc *)
cc[i,k] = c1[i-1,p+k-2] (* instSubscc *)

u[i,k] = a2[i,k-1] (* instSubsu *)
v[k,i] = b2[k-1,i-1] (* instSubsv *)
w[k,i] = c2[k,i-1] (* instSubsw *)

x[k,i] = a3[k-1,i-1] (* instSubsx *)
xx[k,i] = a3[k-1,i+m-2] (* instSubsxx *)
y[i,k:=0] = b3[i-1,0] (* instSubsy *)
yy[i,k:=0] = b3[i+m-2,0] (* instSubsyy *)
z[k,0] = c3[k-1,0] (* instSubsz *)
epsilon = 1
;;

Spaces :
(* assignment of variables to our different spaces in the direct sum *)
a1 b1 c1 (m-1,1,2*p-2) (* m=3,p=3 -> 2 * 1 * 4 = 8 *)
a2 b2 c2 (2,p-1,m-1) (* m=3,p=3 -> 2*2*2 = 8 *)
a3 b3 c3 (p-1,2*m-2,1) (* m=3,p=3 -> 2*4*1 = 8  *)
;;
```

Listing A.1: One of Pan's tables in the input format of our software

which visualized as an expression does not give much intuition:

$$\sum_{i=0}^{(m-1)} \sum_{k=0}^{(p-1)} ((x_{k,i} \cdot \varepsilon^6 + u_{0,k} \cdot \varepsilon^4) + a_{i,0}) \cdot ((y_{i,0} + v_{k,i} \cdot \varepsilon^4) + b_{0,k} \cdot \varepsilon^3) \cdot ((z_{0,k} \cdot \varepsilon^2 + w_{i,0}) + c_{k,i} \cdot \varepsilon^5) +$$
$$\sum_{i=0}^{(m-1)} \sum_{k=0}^{(p-1)} ((xx_{k,i} \cdot \varepsilon^6 + u_{1,k} \cdot \varepsilon^4) - a_{i,0}) \cdot ((yy_{i,0} + v_{k,i} \cdot \varepsilon^4) - bb_{0,k} \cdot \varepsilon^3) \cdot ((z_{0,k} \cdot \varepsilon^2 + w_{i,1}) + cc_{k,i} \cdot \varepsilon^5) +$$
$$\sum_{i=0}^{(m-1)} (-1) \cdot a_{i,0} \cdot ((y_{i,0} + 1/p \cdot \sum_{ev1=1}^{(p-1)} v_{ev1,i} \cdot \varepsilon^4) + 1/p \cdot \sum_{eb=0}^{(p-1)} b_{0,eb} \cdot \varepsilon^3) \cdot (\sum_{ez=1}^{(p-1)} z_{0,ez} \cdot \varepsilon^2 + p \cdot w_{i,0}) +$$
$$\sum_{i=0}^{(m-1)} (-1) \cdot a_{i,0} \cdot ((yy_{i,0} + 1/p \cdot \sum_{ev2=1}^{(p-1)} v_{ev2,i} \cdot \varepsilon^4) + (-1)/p \cdot \sum_{ebb=1}^{(p-1)} bb_{0,ebb} \cdot \varepsilon^3) \cdot (\sum_{ez=1}^{(p-1)} z_{0,ez} \cdot \varepsilon^2 + p \cdot w_{i,1})$$

p	m	N	μ	$s \odot \langle m,n,p \rangle$	Mults	Adds	Total	Naive	ω
12	11	4	$(2,2)$	$6 \odot \langle 121, 12100, 144 \rangle$	310970000	11352994400	11663964400	2521180200	2.780381
13	6	4	$(2,2)$	$6 \odot \langle 36, 3600, 169 \rangle$	38102400	1197532800	1235635200	262051200	2.780250
13	9	4	$(2,2)$	$6 \odot \langle 81, 9216, 169 \rangle$	192264192	6813785088	7006049280	1509414912	2.779593
12	10	4	$(2,2)$	$6 \odot \langle 100, 9801, 144 \rangle$	212789511	7615749438	7828538949	1687732200	2.779192
13	7	4	$(2,2)$	$6 \odot \langle 49, 5184, 169 \rangle$	70574976	2334920256	2405495232	513620352	2.779012
13	8	4	$(2,2)$	$6 \odot \langle 64, 7056, 169 \rangle$	120234240	4134632544	4254866784	913102848	2.778977
12	6	4	$(2,2)$	$6 \odot \langle 36, 3025, 144 \rangle$	27754375	859081850	886836225	187525800	2.778938
12	9	4	$(2,2)$	$6 \odot \langle 81, 7744, 144 \rangle$	139887616	4886448512	5026336128	1080148608	2.778206
11	10	4	$(2,2)$	$6 \odot \langle 100, 8100, 121 \rangle$	150562800	5298485400	5449048200	1171260000	2.777869
11	6	4	$(2,2)$	$6 \odot \langle 36, 2500, 121 \rangle$	19670000	597965000	617635000	130140000	2.777729
12	7	4	$(2,2)$	$6 \odot \langle 49, 4356, 144 \rangle$	51383376	1674777456	1726160832	367550568	2.777644
12	8	4	$(2,2)$	$6 \odot \langle 64, 5929, 144 \rangle$	87506111	2965353776	3052859887	653423232	2.777589
7	6	4	$(2,2)$	$6 \odot \langle 36, 900, 49 \rangle$	3294000	88572600	91866600	18856800	2.776881
11	9	4	$(2,2)$	$6 \odot \langle 81, 6400, 121 \rangle$	99008000	3399891200	3498899200	749606400	2.776877
10	6	4	$(2,2)$	$6 \odot \langle 36, 2025, 100 \rangle$	13492575	401403600	414896175	87042600	2.776714
11	7	4	$(2,2)$	$6 \odot \langle 49, 3600, 121 \rangle$	36396000	1165532400	1201928400	255074400	2.776357
11	8	4	$(2,2)$	$6 \odot \langle 64, 4900, 121 \rangle$	61955600	2063429200	2125384800	453465600	2.776269
9	6	4	$(2,2)$	$6 \odot \langle 36, 1600, 81 \rangle$	8896000	257772800	266668800	55641600	2.776036
8	6	4	$(2,2)$	$6 \odot \langle 36, 1225, 64 \rangle$	5584775	156601550	162186325	33604200	2.775945
10	9	4	$(2,2)$	$6 \odot \langle 81, 5184, 100 \rangle$	67806720	2281208832	2349015552	501365376	2.775672
10	7	4	$(2,2)$	$6 \odot \langle 49, 2916, 100 \rangle$	24949296	782240328	807189624	170603496	2.775229
10	8	4	$(2,2)$	$6 \odot \langle 64, 3969, 100 \rangle$	42448455	1384649154	1427097609	303295104	2.775087
9	7	4	$(2,2)$	$6 \odot \langle 49, 2304, 81 \rangle$	16436736	502209792	518646528	109057536	2.774391
9	8	4	$(2,2)$	$6 \odot \langle 64, 3136, 81 \rangle$	27948032	888798848	916746880	193880064	2.774163
8	7	4	$(2,2)$	$6 \odot \langle 49, 1764, 64 \rangle$	10308816	305002656	315311472	65864232	2.774065

Table A.1: Best Complexities of Schönhage's tensor with parameters m and p varying from 2 to 14 and tensor power from 2 to 4.

p	m	N	μ	$s \odot \langle m,n,p \rangle$	Mults	Adds	Total	Naive	ω
12	8	3	$(1,1,1)$	$6 \odot \langle 154,154,154 \rangle$	8630776	287031745	295662521	43684872	2.815013
14	7	3	$(1,1,1)$	$6 \odot \langle 156,156,156 \rangle$	8942544	298502880	307445424	45410976	2.814847
15	7	3	$(1,1,1)$	$6 \odot \langle 168,168,168 \rangle$	10890432	370450080	381340512	56730240	2.812595
11	10	3	$(1,1,1)$	$6 \odot \langle 180,180,180 \rangle$	13214880	452834100	466048980	69789600	2.812481
13	8	3	$(1,1,1)$	$6 \odot \langle 168,168,168 \rangle$	10833984	369437376	380271360	56730240	2.811581
12	9	3	$(1,1,1)$	$6 \odot \langle 176,176,176 \rangle$	12309440	423127584	435437024	65235456	2.810978
14	8	3	$(1,1,1)$	$6 \odot \langle 182,182,182 \rangle$	13379912	466261705	479641617	72144072	2.808894
12	10	3	$(1,1,1)$	$6 \odot \langle 198,198,198 \rangle$	16905240	596599047	613504287	92913480	2.808363
13	9	3	$(1,1,1)$	$6 \odot \langle 192,192,192 \rangle$	15450624	544627968	560078592	84713472	2.807687
15	8	3	$(1,1,1)$	$6 \odot \langle 196,196,196 \rangle$	16293088	578666284	594959372	90123936	2.806777
12	11	3	$(1,1,1)$	$6 \odot \langle 220,220,220 \rangle$	22520080	811980400	834500480	127485600	2.806675
13	10	3	$(1,1,1)$	$6 \odot \langle 216,216,216 \rangle$	21218112	767936160	789154272	120652416	2.805177
14	9	3	$(1,1,1)$	$6 \odot \langle 208,208,208 \rangle$	19080256	687391328	706471584	107727360	2.805114
13	11	3	$(1,1,1)$	$6 \odot \langle 240,240,240 \rangle$	28264320	1045200000	1073464320	165542400	2.803569
15	9	3	$(1,1,1)$	$6 \odot \langle 224,224,224 \rangle$	23233280	853130880	876364160	134572032	2.803091
14	10	3	$(1,1,1)$	$6 \odot \langle 234,234,234 \rangle$	26201448	969262047	995463495	153426312	2.802689
13	12	3	$(1,1,1)$	$6 \odot \langle 264,264,264 \rangle$	36717120	1382257536	1418974656	220378752	2.802571
14	11	3	$(1,1,1)$	$6 \odot \langle 260,260,260 \rangle$	34901360	1319244160	1354145520	210506400	2.801145
15	10	3	$(1,1,1)$	$6 \odot \langle 252,252,252 \rangle$	31903200	1202993316	1234896516	191655072	2.800734
14	12	3	$(1,1,1)$	$6 \odot \langle 286,286,286 \rangle$	45337864	1744707965	1790045829	280233096	2.800197
14	13	3	$(1,1,1)$	$6 \odot \langle 312,312,312 \rangle$	57668832	2253023760	2310692592	363871872	2.799662
15	11	3	$(1,1,1)$	$6 \odot \langle 280,280,280 \rangle$	42495040	1637403040	1679898080	262953600	2.799242
15	12	3	$(1,1,1)$	$6 \odot \langle 308,308,308 \rangle$	55200992	2165509500	2220710492	350048160	2.798334
15	13	3	$(1,1,1)$	$6 \odot \langle 336,336,336 \rangle$	70213248	2796462144	2866675392	454519296	2.797830
15	14	3	$(1,1,1)$	$6 \odot \langle 364,364,364 \rangle$	87724000	3539410420	3627134420	577947552	2.797612

Table A.2: Best Complexities of Pan's tensor (presented in Section 4.4.3) with parameters m and p varying from 2 to 15 and tensor power from 2 to 3.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)
université PARIS-SACLAY

Titre : Investigations en Mathématiques Assistées par Ordinateur: Expérimentation, Calcul et Certification

Mots clefs : Preuve formelle, Calcul numérique, Calcul formel

Résumé : Cette thèse présente trois contributions au thème des preuves assistées par ordinateur: il s'agit de preuves reposant sur le calcul et de preuves formelles produites et vérifiées à l'aide d'un logiciel appelé assistant à la preuve.
Nous illustrons d'abord le thème de l'expérimentation au service de la preuve, en programmant un outil de manipulation symbolique et en l'utilisant pour trouver et démontrer un résultat concernant les produits de matrices.
Dans une seconde partie, nous utilisons un assistant à la preuve pour écrire une démonstration du théorème d'Apéry, qui repose sur des calculs complexes. Ces calculs sont effectués par un programme externe efficace; ils sont ensuite vérifiés par ce logiciel de preuve.
Dans la troisième contribution, nous proposons un programme qui calcule des intégrales et construit simultanément une preuve que le résultat est correct. Nous utilisons ce programme pour trouver des erreurs dans des résultats publiés.

Title : Investigations in Computer-Aided Mathematics: Experimentation, Computation, and Certification

Keywords : Formal proof, Numeric Computations, Computer Algebra

Abstract : This thesis presents three contributions to the topic of computer-assisted proofs: both in the case of proofs relying on computations and of formal proofs produced and verified using a piece of software called a proof assistant.
We first illustrate the theme of experimentation at the service of proofs by programming a symbolic manipulation tool and using it to find and demonstrate a result about matrix multiplications.
In a second part, we use a proof assistant to write a proof of Apéry's theorem, which is based on complex computations. These computations are performed by an efficient external program and are then verified by this proof software.
In the third contribution, we propose a program which computes integrals and simultaneously builds proof that the result is correct. We use this program to find errors in published results.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France