

Outils pour la Formalisation en Analyse Classique

Une Étude de Cas en Théorie du Contrôle

Damien Rouhling

Inria Sophia Antipolis Méditerranée

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur.

Dirigée par : Yves Bertot.

Co-encadrée par : Cyril Cohen.

Soutenue le : TBA.

Devant le jury, composé de :

Jesús Aransay
Yves Bertot
Sylvie Boldo
Cyril Cohen
Éric Goubault
Étienne Lozes
Lawrence Paulson

Outils pour la Formalisation en Analyse Classique

Une Étude de Cas en Théorie du Contrôle

Jury :

Directeurs

Yves Bertot, Directeur de Recherche, Inria Sophia Antipolis Méditerranée
Cyril Cohen, Chargé de Recherche, Inria Sophia Antipolis Méditerranée

Rapporteurs

Sylvie Boldo, Directrice de Recherche, Inria Saclay Île de France
Lawrence Paulson, Professor, University of Cambridge

Examinateurs

Jesús Aransay, Profesor Contratado Doctor, Universida de La Rioja
Éric Goubault, Professeur, École Polytechnique
Étienne Lozes, Professeur, Université de Nice Sophia Antipolis

Formalisation Tools for Classical Analysis

A Case Study in Control Theory

Jury:

Advisors

Yves Bertot, Directeur de Recherche, Inria Sophia Antipolis Méditerranée
Cyril Cohen, Chargé de Recherche, Inria Sophia Antipolis Méditerranée

Rapporteurs

Sylvie Boldo, Directrice de Recherche, Inria Saclay Île de France
Lawrence Paulson, Professor, University of Cambridge

Examiners

Jesús Aransay, Profesor Contratado Doctor, Universida de La Rioja
Éric Goubault, Professeur, École Polytechnique
Étienne Lozes, Professeur, Université de Nice Sophia Antipolis

Outils pour la Formalisation en Analyse Classique

Une Étude de Cas en Théorie du Contrôle

Résumé :

Il s'agit de mettre à l'épreuve une bibliothèque d'analyse dans l'assistant de preuve Coq au travers d'une étude de cas en théorie du contrôle. Nous formalisons une preuve de stabilité pour le pendule inversé, un exemple classique en théorie du contrôle. Le contrôle du pendule inversé est un défi en raison de sa non-linéarité, à tel point que ce système est souvent utilisé comme référence pour l'essai de nouvelles techniques de contrôle.

Durant cette étude de cas, nous identifions des défauts des outils aujourd'hui accessibles pour la formalisation en analyse classique et nous en développons d'autres afin d'atteindre le but de cette étude de cas. En particulier, nous essayons d'imiter le style de preuve sur papier grâce à de nouvelles notations et de nouveaux mécanismes d'inférence. C'est une étape essentielle pour rendre la preuve formelle plus accessible aux mathématiciens.

Ensuite, nous développons une nouvelle bibliothèque d'analyse classique en Coq, qui intègre ces nouveaux outils et qui essaie de pallier les limitations de la bibliothèque que nous avons testée, en particulier dans le domaine du raisonnement asymptotique. Nous testons aussi cette nouvelle bibliothèque sur la même preuve formelle et tirons des conclusions sur ses forces et faiblesses.

Enfin, nous esquissons une nouvelle méthodologie pour répondre aux limitations de notre bibliothèque dans le domaine du calcul. Nous exploitons une technique appelée raffinement afin de refactoriser la méthode de preuve par réflexion, une technique qui automatise les preuves grâce au calcul et qui de plus réduit la taille des termes de preuves. Nous mettons en œuvre cette méthodologie sur l'exemple du raisonnement arithmétique dans les anneaux et expliquons comment ce travail pourrait servir à généraliser des outils déjà existants.

Mots clés : Preuve formelle, Coq, analyse classique, théorie du contrôle, pendule inversé, automatisation, calcul.

Formalisation Tools for Classical Analysis

A Case Study in Control Theory

Abstract:

In this thesis, we put a library for analysis in the Coq proof assistant to the test through a case study in control theory. We formalise a proof of stability for the inverted pendulum, a standard example in control theory. Controlling the inverted pendulum is challenging because of its non-linearity, so that this system is often used as a benchmark for new control techniques.

Through this case study, we identify issues in the tools that are currently available for the formalisation of classical analysis and we develop new ones in order to achieve our formalisation goal. In particular, we try to imitate the pen-and-paper proof style thanks to new notations and inference mechanisms. This is an essential step to make formal proofs more accessible to mathematicians.

We then develop a new library for classical analysis in Coq that integrates these new tools and tries to palliate the limitations of the library we tested, especially in the domain of asymptotic reasoning. We also experiment with this new library on the same formal proof and draw lessons on its strengths and weaknesses.

Finally, we sketch a new methodology in order to address the limitations of our library in the particular domain of computation. We exploit a technique called refinement to refactor the methodology of proof by reflection, a technique that automates proofs through computation and also reduces the size of proof terms. We implement this methodology on the example of arithmetic reasoning in rings and discuss how this work could be used to generalise existing tools.

Keywords: Formal proof, Coq, classical analysis, control theory, inverted pendulum, automation, computation.

CONTENTS

Introduction **xv**

I Case Study: the Inverted Pendulum **1**

1 Context **3**
1.1 The Inverted Pendulum . . . 3
1.1.1 The System . . . 3
1.1.2 Notions in Control Theory and Application to the Pendulum . . . 4
1.1.3 Control of the Inverted Pendulum . . . 6
1.2 Modelling Physical Systems . . . 6
1.2.1 On Dynamical Systems . . . 7
1.2.2 Taking into Account the Control Function . . . 8
1.2.3 An Important Property: Stability . . . 10
1.3 Practical Aspects of such a Study . . . 11
1.3.1 How to Formally Study a Physical System . . . 11
1.3.2 Scope of our Case Study . . . 13

2 LaSalle's Invariance Principle **15**
2.1 The Original Principle . . . 15
2.1.1 Intuition behind the Invariance Principle . . . 16
2.1.2 Statement of LaSalle's Invariance Principle . . . 17
2.1.3 Proof of the Invariance Principle . . . 19
2.2 Generalisation of LaSalle's Invariance Principle . . . 21
2.2.1 Weaker Hypotheses for the Invariance Principle . . . 21
2.2.2 A Stronger Invariance Principle . . . 23
2.3 Formalisation of the Generalised Principle . . . 24
2.3.1 A Note on Logical Foundations and Choosing a Library . . . 24
2.3.2 Filters for Real Analysis . . . 27

2.3.3 Topological Notions . . . 32
2.3.4 Formal Statement of the Invariance Principle . . . 36
2.4 Related Work . . . 39
2.4.1 Related Work on Stability Analysis . . . 39
2.4.2 Related Work on the Formalisation of Topology . . . 40
2.4.3 Related Work on the Formalisation of Differential Equations . . . 40

3 Swing-Up of the Inverted Pendulum **43**
3.1 The Dynamical System . . . 43
3.1.1 The Dynamical System and its Control Challenge . . . 44
3.1.2 The System We Actually Formalised . . . 45
3.2 Stability Proof . . . 48
3.2.1 Verification of the Hypotheses of LaSalle's Invariance Principle . . . 48
3.2.2 Convergence to the Homoclinic Orbit . . . 50
3.2.3 Summary of the Corrected Errors . . . 53
3.3 Formalisation of the Stability Proof . . . 53
3.3.1 On the Choice of Data Structures . . . 53
3.3.2 Topological Spaces . . . 55
3.3.3 Automatic Computation of Differentials . . . 57
3.4 Related Work . . . 60
3.4.1 Related Work on Dynamical Systems and Control Theory . . . 60
3.4.2 Related Work on the Formalisation of Mathematics . . . 61

4 Assessment of the Formalisation **63**
4.1 Improvements on the Existing . . . 63
4.1.1 A Smoother Experience with Coquelicot . . . 63
4.1.2 Using Coquelicot in other Fields of Mathematics . . . 64
4.2 Possible Extensions . . . 64
4.2.1 Completing the Proof of Stability . . . 65
4.2.2 Towards a Certified Implementation . . . 65
4.3 Remaining Complications . . . 66
4.3.1 Discrepancy with Pen-and-Paper Mathematics . . . 66
4.3.2 Missing Tools . . . 67
4.3.3 Combining Several Hierarchies . . . 67

II Designing a Library of Mathematics **69**

5 Hierarchy of the Mathematical Components Analysis Library **71**
5.1 Principles of Design . . . 71
5.1.1 Logical Foundations . . . 72
5.1.2 Organising the Library . . . 73
5.2 The Starting Point: Coquelicot . . . 75
5.2.1 Coquelicot's Hierarchy . . . 75
5.2.2 Making Coquelicot Compatible with Mathematical Components 77
5.2.3 Minor Improvements in the Hierarchy . . . 79

5.3 Extension of the Hierarchy . . . 81
5.3.1 Topological Spaces . . . 81
5.3.2 Filtered Spaces . . . 82
5.3.3 Non-Empty Spaces . . . 84
5.4 Modification of the Interfaces . . . 87
5.4.1 Refactoring Normed Spaces . . . 87
5.4.2 A More Abstact Definition of Uniform Spaces . . . 88
5.4.3 Removing the Dependency on the Standard Library . . . 89

6 Tools for Asymptotic Reasoning 91
6.1 Small-Scale Filter Elimination . . . 92
6.1.1 The `near` Tactics . . . 92
6.1.2 Example: a Short Completeness Proof . . . 95
6.2 Bachmann-Landau Notations . . . 99
6.2.1 Mechanisation of Equational Bachmann-Landau Notations . . . 99
6.2.2 Examples and Applications . . . 102

7 Evaluation of our Library 107
7.1 Improvements on our Case Study . . . 107
7.2 Remaining and New Issues . . . 108
7.3 Related Work . . . 109
7.3.1 Libraries for Analysis . . . 109
7.3.2 Related Work on Asymptotic Reasoning . . . 110
7.3.3 Related Work on Delayed Production of Witnesses . . . 111

III Tools for Automation 113

8 Refinement and Computation 115
8.1 Refinement . . . 115
8.1.1 Definition of Refinement . . . 116
8.1.2 Program Refinement . . . 117
8.1.3 Data Refinement . . . 118
8.1.4 Composition of Refinements . . . 119
8.2 Using Refinement in Proofs . . . 120
8.2.1 On Proofs and Computation . . . 120
8.2.2 Automation of Refinement . . . 123
8.2.3 A Simplification Tactic . . . 125
8.3 The Benefits of Parametricity . . . 127
8.3.1 The Parametricity Theorem . . . 127
8.3.2 Parametricity for Data Refinement . . . 129
8.3.3 Current and Future Work on Parametricity . . . 130

9 Proof by Reflection **133**
9.1 Principles of Proof by Reflection 133
9.2 A More Modular Methodology 136
9.3 Possible Improvements and Future Work 139
9.3.1 Missing Features 139
9.3.2 Efficiency issues 140
9.3.3 Possible Generalisations 140

Conclusion **143**

List of Figures **147**

Bibliography **149**

INTRODUCTION

On Formal Proofs

Computer programs are playing a predominant role in our society. We entrust them with our communications (email agents, web browsers, radio software), with our economy (automated assembly lines, real-time trading tools), and even with our lives (robotic surgery, autopilot, military robots). With such a wide range of applications, many of them being critical, it is crucial to ensure that no error sneaked in these programs. These errors can be of two kinds: they can come from the actual implementation of the program or they can originate from the design of the underlying algorithm.

Implementation errors seem less important because they may be easier to spot. One can indeed notice some of them at compilation time, especially if the program is written using a strongly typed language, preventing even the execution of the program. However, most of them pass the compilation step so that it is necessary to resort to more complex techniques to spot them. One of these techniques, the most natural one, is testing. Running the program and checking its output against the expected one may reveal a bad behaviour. In order to locate its origin, observing the output is often not sufficient and one has to check the whole execution trace.

In this way, testing may also expose design errors. Imagine for instance a program whose result depends on one particular computation, which is the implementation of an erroneous mathematical formula. The execution trace may reveal a wrong result for this computation and make the programmer check this formula. However, the execution trace will not reveal whether an error stems from an erroneous formula or from the implementation of a valid formula, e.g. from approximations resulting from the necessity of representing real numbers with a finite precision.

In spite of it being widely used, testing is no magic bullet. According to Dijkstra [Dij72], "program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness". Let me emphasise the meaning of this last sentence. In order to bring guarantees on a computer program, one has to

prove its correctness. However, **even proofs can be erroneous**. That is why a correctness proof must be **convincing** so that one can be **confident** in the result of the program.

Is confidence sufficient? Shall we consider using computer programs only in non-critical domains of application? Our society has already decided that the benefits exceed by far the risks incurred, but not without imposing a few limitations. The necessity for the level of confidence to match the level of safety required for an application already appears in software safety standards for avionics. In particular, *formal methods* must be used in complement of testing for the verification of parts of the system [RTC11]. "Formal methods" refers to a set of techniques for the verification of a *specification*. A specification is a mathematical property, expressed in a given logic, that a system has to satisfy. It is the description of the expected behaviour of a program, its safety condition. The main characteristics of a specification is that it is symbolic. The chosen logic defines how each symbol can be manipulated and the goal of formal verification is to provide a proof of the specification which is valid according to the rules of manipulation of the symbols.

The main benefit of symbolic logic is that we can mechanise the process of verification, i.e. we can use computer programs either to build a proof of the specification or to check that this proof indeed respects the rules of the logic. Isn't the use of computer programs to check the correctness of other programs like a dog chasing its own tail? These verifiers can contain errors, too. What could possibly happen if a faulty verifier produced/accepted an erroneous proof of correctness of an unsound program? In fact, this is no issue, for everything boils down to the level of confidence one desire for their program. This is "a matter of trust", as phrased by Keller [Kel13]. I would personally rather trust someone who gives me a machine-checked proof than "only" a pen-and-paper proof.

This thesis is concerned with the domain of formal methods that deals with the computer-aided verification of proofs built by a human being: *interactive theorem proving*. Interactive theorem provers, also known as *proof assistants*, are programs that check that both a statement and its proof, input by a user, are well-formed. This means that the burden of finding a proof falls to the user and that the proof assistant deals with a task that is easier for a computer: mechanically applying rules to check the proof. This does not sound appealing, since the goal of mechanising a process is to free human beings from "hard" tasks. But in fact, this has several advantages. First, simple tasks make for simple programs. As a consequence, the source code of a proof assistant will be easier to read, so that we can convince ourselves that it is correct. More precisely, since a proof assistant cannot just be reduced to a proof checker, only the part that indeed checks the proofs, the kernel, has to be simple and small, ideally readable by a human being. This is called the de Bruijn criterion, as coined by Barendregt and Barendsen [BB02]. On top of the kernel, proof assistants include various facilities in order to smoothen the user experience. Although they may deceive the users into thinking they proved a false result [Wie12], they do not need to be correct since they send to the kernel each proof that has to be checked.

Then, another advantage that stems from the user having to provide a proof is that we can make the most of human cleverness. Indeed, many of the involved decision problems either have a high computational complexity (e.g. Presburger arithmetic [FR98]) or are even undecidable [Mon76]. That is why modern automated provers use various heuristics to speed up the proof search on given families of logical formulas, and to maximise the number of such families. Instead of trying to have computer programs make clever choices, in interactive

theorem proving the users have to make them themselves. Thus, one has a better control on how the proof is led and can avoid logical dead ends by performing the appropriate reasoning steps.

Moreover, the process of building a proof can be beneficial for the user. Instead of throwing a formula at an automated prover and keeping their fingers crossed, thinking hard to find why this formula holds gives insights into the manipulated objects. This is all the more patent with interactive proofs: pen-and-paper proofs contain holes, such as the implicit use of different equivalent definitions of the same mathematical object, while a proof assistant requires its user to explicitly fill them in. This means thinking about how to state each property, which definition to use for each notion, in order to be in an optimal context for proofs. To this end, it is easier to start with a pen-and-paper proof and then to implement it inside a proof assistant.

Motivations of this Work

Although writing a proof on paper and then verifying it with a proof assistant is presently the most efficient way to do interactive theorem proving, this is not entirely satisfactory. Often, unexpected issues arise when one starts the implementation part: the tools at hand do not use the same representation of the objects as the one required for the proof, which sometimes means code duplication [Ben06], one has to adapt the mathematical objects to the finite world of computers (e.g. working with truncated formal power series in order to be able to decide equality [Dja18]), or even the proof is erroneous and one may have to start again (at least partly) from scratch (see Section 3.2 for instance).

In an ideal world, proof assistants would be used for mathematical exploration, the computer checking the small details while the mathematician is testing new ideas. This idea already appears in Simpson's plea for the application of formal proofs to mathematics [Sim04]. This is also true for the verification of programs: as Dijkstra advocates [Dij72], "one should not first make the program and then prove its correctness [...]. On the contrary: the programmer should let correctness proof and program grow hand in hand". Formal methods should then be integrated into the development process.

Several issues however impede progress in this direction. First of all, interactive theorem proving is not easy. As Benton says [Ben06], "it's a strange new skill, much harder to learn than a new programming language or application, or even many bits of mathematics". Efforts need to be done in order for proof assistants to be more friendly to newcomers, especially to mathematicians: these tools are often developed with concerns about logic and programming in mind [Sim04], which are quite different to mathematical concerns.

Another issue of interactive theorem proving is the difficulty to reach the appropriate level of automation. Working with a proof assistant is a tedious process: proofs are verbose because the computer requires the user to show every single statement, even the most trivial ones like $0 \leqslant 1$. Full automation also has its own drawbacks [Ben13]. Modern proof assistants include inference mechanisms but knowing which one to use, and how and when to use it already requires experience.

Even with proper tools for automation, an additional limitation is the amount of background theory that has to be developed. Simpson's description of topics of mathematics in which formalisations were done or are to be done [Sim04] is quite impressive by its length,

even more if we take into consideration the time required to achieve these formalisations. The already achieved formalisations are spread in different libraries (the Archive of Formal Proofs [BHMN15], Coquelicot [BLM15], CoRN [CGW04], the Lean Mathematical Components library [LMCLD], Mathematical Components [MCT], the Mizar library [BBG+18] are only a few of them), for different proof assistants (Coq [CDT19], HOL Light [Har16], Isabelle/HOL [NPW02], Lean [dMKA+15], Mizar [NK09], PVS [ORS92] and others [Wie06]), which work in different logical settings. Moreover, some libraries have their own limitations, which drive users to redevelop parts of them in a different way.

Efforts have been made to overcome these obstacles. In front of the insufficiency of reference manuals, books were written in order to give clues on how to use different systems [BC04, MT18, AdMK18], sometimes with a focus on a particular domain of application of the system [BM17]. Automation is a vast research topic, so I will cite only a few pieces of work. Inference mechanisms have been developed, e.g. type classes in Isabelle/HOL [HW06], Coq [SO08] and Lean [dMKA+15] and canonical structures in Coq [Sai99, MT13]. These mechanisms have various applications, not only in proof automation [GZND11], but also in the design of libraries, e.g. via the formalisation of a hierarchy of mathematical structures for algebra [Coh12, GAA+13] or for analysis [HIH13, BLM15]. Computation is also an important aspect of automation. Its use has been developed through mechanisms such as reflection [Bou97] and refinement [DMS12, Lam13]. Finally, the communication of proof assistants with automated procedures, such as SAT/SMT solvers [FMM+06, AFG+11, Kel13] or automated theorem provers [MBG06], and the communication between different proof assistants [KW10, Kel13, CD17] have been explored.

This thesis falls in with these efforts to provide proof assistants with well-adapted tools and libraries in order to make interactive theorem proving easier. In my work, I focused on one particular proof assistant: Coq. Before stating my contributions to interactive proving with Coq, let me introduce this proof assistant and give my motivations for working with it.

The Coq Proof Assistant



Warning

Throughout this thesis, we try to keep code snippets simple and as relevant as possible with respect to the discussion. As a consequence, some of them are different from the actual implementation: some parts may be reshaped, renamed or omitted.

Key Features

Coq [CDT19] is a proof assistant based on *type theory*, i.e. on a typed λ-calculus. Type theories make it possible to describe programs, properties and proofs **in the same language**. Indeed, λ-calculus is in itself a programming language, though not very practical. Moreover, propositions can be described by types and proofs of these propositions by λ-terms [How80]: this is called the Curry-Howard correspondence.

Coq's type system, called the Calculus of Inductive Constructions, extends the Calculus of Constructions [Coq85, CH88] with *inductive types* [CP88]. An important feature of these

calculi is that they include *dependent types*, i.e. types that can depend on arbitrary expressions. Combined with inductive types, dependent types make Coq's language very expressive. For instance, Coq's standard library contains a type of vectors, i.e. a data structure of lists that contains the information of their length.

```
Inductive vector (A : Type) : nat -> Type :=
  | nil : vector A 0
  | cons : forall (h : A) (n : nat), vector A n -> vector A (S n).
```



Remark

The `vector` data type is also *polymorphic*: a single definition of the type makes it possible to handle both vectors of integers and vectors of real numbers.

With this language, one can write programs that embed (parts of) their specification: preconditions (respectively postconditions) can be described by the type of the inputs (respectively outputs) of the program. This is a very strong interpretation of Dijkstra's recommendation of "[letting] correctness proof and program grow hand in hand" [Dij72]. An example of partial specification by definition is the `append` function on vectors, which already contains the information that the length of its output is the sum of the lengths of its inputs.

```
Fixpoint append (A : Type) (n p : nat) (v : vector A n)
  (w : vector A p) : vector A (n + p) :=
  match v with
  | nil _ => w
  | cons a v' => cons a (append v' w)
  end.
```



Remark

Coq comes with a type inference mechanism, which allows the user to let arguments unspecified and even to write code as if these arguments did not exist: they are called *implicit arguments*. For instance, the arguments `A` and `n` of the `cons` function are implicit and do not appear in the definition of `append`, since they can be inferred (e.g. from the types of `a` and `v'` in its first use).

From now on we will follow Coq's syntax for implicit arguments. When giving a new definition, implicit arguments are declared using curly brackets, as in

```
Definition fct {arg1 : type1} (arg2 : type2) := ... .
```

Then, $\texttt{arg}_1$ is omitted in subsequent uses of `fct`, which all are of the form $\texttt{fct arg}_2$. Still, it is possible to give implicit arguments explicitly, using the `@` symbol before the name of the function: $\texttt{@fct arg}_1\ \texttt{arg}_2$.

The `append` function also illustrates the need for another feature of Coq: *conversion*. Indeed, imagine one wants to prove that appending the vectors [0] and [1] gives the vector [0; 1]. In Coq, one would write

```
Lemma app_01 :
  append (cons 0 (nil nat)) (cons 1 (nil nat)) = cons 0 (cons 1 (nil nat)).
```

However, one can only compare two terms of the same type, and the left-hand side of the equation has type `vector nat (1 + 1)`, while the right-hand side has type `vector nat 2`. This is where conversion comes into play. In the Calculus of Inductive Constructions, two terms are said convertible if they are related through a particular equivalence relation $\equiv$, which tries to capture the notion of equality by computation. The following conversion rule states that if t is a term of type T and U is well-formed (i.e. U has a sort s as type), and if T and U are convertible, then t also has type U [1].

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \equiv U}{\Gamma \vdash t : U}$$

Since `1 + 1` and `2` are convertible, `vector nat (1 + 1)` and `vector nat 2` are also convertible and, thanks to the conversion rule, Lemma `app_01` is indeed well-formed.



Remark

The two sides of the equation in Lemma `app_01` are convertible, hence the reflexivity of equality is sufficient to prove this lemma.

From an implementation point of view, Coq respects the de Bruijn criterion: it only relies for its correctness on a kernel that checks the proofs. Since Coq is based on type theory, checking the proofs means here checking that a given λ-term has a given type. This is the role of Coq's type-checker.

Finally, a last feature of Coq I would like to mention, also related to its implementation, is its *proof mode*. In order to prove a proposition, the user of Coq does not have to provide a λ-term which inhabits the type describing this proposition. Instead, the user can progressively build it using *tactics* (see seminal work by Gordon et al. [GMW79] and Milner [Mil85], and Delahaye's work in the particular case of Coq [Del00]). The proposition to prove is called a *goal*. Coq also displays a *context*, which contains the variables and hypotheses that can be used to prove the goal. With tactics, the user apply transformations to the goal and to the context, possibly generating new goals. Internally, the tactics build a λ-term with holes, which are filled in once each goal is closed.



Remark

There is an extension of Coq's tactic language, SSReflect [GMT15], which we used for this work.

1. In fact, in the actual implementation of Coq, the convertibility condition $T \equiv U$ is replaced with a subtyping condition $T \leqslant U$ [CDT19].



Example

Let me illustrate Coq's proof mode on a very simple example: the proof of the commutativity of the addition on natural numbers.

```
Lemma addnC (n m : nat) : n + m = m + n.
```

The proof script starts with the `Proof` command to mark the beginning of the proof. Then we proceed by induction on `n` through the `elim: n` tactic. At this point, Coq shows two goals (corresponding to the base case and the induction step) and the context of the first one.

```
2 subgoals

  m : nat
  ============================
  0 + m = m + 0

subgoal 2 is:
 forall n : nat, n + m = m + n -> n.+1 + m = m + n.+1
```

So, using only `m` and already proven lemmas, we have to prove the first equation. Conversion reduces the left-hand side of the equation to `m` but not the right-hand side. It is necessary to rewrite with Lemma `addn0` to do so. Thus, we can close this subgoal by typing `by rewrite addn0`. Coq then prints the second subgoal with its context.

```
  m : nat
  ============================
  forall n : nat, n + m = m + n -> n.+1 + m = m + n.+1
```

We can then put `n` together with the induction hypothesis in the context using the `move=> n ihn` tactic.

```
  m, n : nat
  ihn : n + m = m + n
  ============================
  n.+1 + m = m + n.+1
```

The next step is to replace the right-hand side of the equation with `(m + n).+1` by rewriting with Lemma `addnS` and then to use `ihn` from right to left in order to swap `m` and `n` in the resulting expression. We thus type `rewrite addnS -ihn` and Coq prints:

```
  m, n : nat
  ihn : n + m = m + n
  ============================
  n.+1 + m = (n + m).+1
```

This is true by computation and after the `by []` tactic Coq prints:

```
No more subgoals.
```

We can then register this lemma for later use with the `Qed` command.

The advantage of such a proof mode becomes clear when one compares this script, which can be condensed into the one-liner proof below and which is a description of the sequence of reasoning steps, and the corresponding ($\sim$ 10 lines) λ-term obtained through the `Print addnC` command, which is harder to read and contains irrelevant information (although proof scripts are also inherently hard to read [Ben06]).

```
Proof. by elim: n => [|n ihn]; rewrite ?addn0 // addnS -ihn. Qed.
```

Why Coq?

Most of these features are not specific to Coq, so let me explain my reasons to focus on this proof assistant in this thesis.

First, the most personal one: I already knew Coq before starting to work on this thesis. This is clearly not a sufficient reason in itself, since I could have learnt to use another proof assistant. But, the gain of time put aside, this has an advantage: I already had an idea of what was hard/tedious in Coq, of which domains of mathematics were insufficiently tackled in Coq.

Then, type theory in general and the Calculus of Inductive Constructions in particular provide a language that constitutes a unified framework for writing programs, theorems and proofs. As a consequence, as Benton puts it [Ben13], "a proof assistant like Coq is in many respects simply a better programming language than most conventional ones".

Since Coq is not the only proof assistant based on type theory, nor on the Calculus of Inductive Constructions, there is a third reason to focus on it, which is more social. Coq has a large community of users, and contacts with industry through the Coq Consortium. This sets up a good environment for a system that constantly evolves and adapts to the needs of its users.

This brings us to my last argument for using Coq: throughout the years, its large community has built tools and well-furnished libraries. This makes both for a great toolkit to work with and for a working base for our study. I will only cite the Mathematical Components library [MCT], based on the SSReflect extension of Coq's tactic language [GMT15], and the Coquelicot library [BLM15], which both played a significant role in this thesis.

Contributions

Making interactive theorem proving in Coq easier is a huge programme, so I had to make a choice. I could have worked on Coq's implementation in order to improve the internal behaviour of some tools or come up with new ideas on its internals, but I chose instead to leave the proof assistant untouched and to work on top of it. Developing formalisation tools only makes sense if these tools have some use. It is then important to put these tools to the test with concrete examples. Having a penchant for mathematics, especially for analysis, I decided to explore the use of Coq in this domain. Since libraries for analysis already exist in Coq (see Section 2.3.1), I developed a case study to determine (and palliate) their limitations.

For this experiment, I focused on a concrete example in *control theory*. This field deals with dynamical systems, which usually operate in continuous time. This makes control theory a good candidate to study formalisation techniques in analysis. A goal of this field is to design

techniques to grant a particular behaviour of a dynamical system: a *control function* affects the reaction of the system to its inputs in order to achieve this goal.

My case study consists of a formal proof of correctness of a given control function for a standard example in control theory: the inverted pendulum (see Section 1.1 for a description of the pendulum and of the control challenges associated to this system). I started from a pen-and-paper proof [LFB00] and implemented it in Coq. This proof goes through the use of LaSalle's invariance principle [LaS60, LaS76], which is widely used in the analysis of the stability of dynamical systems (see Section 1.2.3 for an introduction to stability analysis). I formalised in Coq a generalised version of this principle.

This case study required the development of several tools for analysis. I formalised several notions of topology, up to the proof of Tychonoff's Theorem. Cyril Cohen and I designed notations that, combined with a filter inference mechanism, make easier the manipulation of limits. I worked on combining different libraries (namely Coquelicot [BLM15] and Mathematical Components [MCT]), especially to obtain a ready to use formalisation of $\mathbb{R}^n$. Finally, I also developed a mechanism to compute automatically the derivative/differential of a function.

From this experiment, we were convinced that a new framework for analysis in Coq was needed. Reynald Affeldt, Cyril Cohen and I started to work on a fork of Coquelicot, which rapidly mutated into a whole new library, Mathematical Components Analysis [ACM+], compatible with Mathematical Components by design. We also collaborate with Assia Mahboubi and Pierre-Yves Strub on this library since it now depends on a new description of real numbers that they developed before this library even existed. The parts of the case study that could be used in a more general context were added to the library. We also developed new tools for asymptotic reasoning. My other contributions to this library consist of the proof of standard theorems (e.g. Zorn's Lemma, Heine-Borel's Theorem, the Intermediate Value Theorem, Rolle's Theorem, and the Mean Value Theorem), different modifications of the topological hierarchy, the adaptation of the library to the new description of real numbers, and most of the documentation.

I also worked on tools for the automation of reasoning, that were missing from the new library. I focused on proofs by computation and the proof technique called *reflection*. Cyril Cohen and I designed a more modular methodology for reflection and applied it to a prototype of tactic to decide equalities on arithmetic expressions, similar to `ring` [GM05].

Publications

The case study was published in two parts. First, the formalisation of LaSalle's invariance principle appeared in the proceedings of the 8th International Conference on Interactive Theorem Proving [CR17a]. Then, its application to the inverted pendulum appeared in the proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs [Rou18]. Both formalisations may be found on-line in the same repository [CRa].

The Mathematical Components Analysis library was first introduced at the Coq Workshop 2018 [ACM+18]. A more technical presentation of some tools we designed for the library was published in the Journal of Formalized Reasoning [ACR18]. The code of the library is freely accessible on-line [ACM+].

Our methodology for proofs by reflection and the prototype of tactic based on this methodology appeared in the proceedings of the 28e Journées Francophones des Langages Applicat-

ifs [CR17b], which is a French-speaking conference. The prototype of tactic is published as a part of the CoqEAL library [CCD+].

Organisation of this Thesis

This thesis is organised in three parts which do not respect chronological order. Part III was in fact the starting point of my work, as a continuation of my Master's internship, but it contains hints of solutions to some issues raised in the remainder of this thesis. Hence, these issues are presented first.

Part I describes the case study.

Chapter 1 gives the necessary context to understand this experiment. It introduces the inverted pendulum together with its control challenges, defines the notions that are important to model the pendulum as a dynamical system, and contains my thoughts and choices on how to formalise such a system.

Chapter 2 and Chapter 3 respectively describe the formalisations of LaSalle's invariance principle and of the inverted pendulum. They are both organised as follows: I first explain the mathematics involved in the proofs and then present the tools I formalised to implement these proofs in Coq.

Chapter 4 gives an assessment of this formalisation and explains why we started working on a new library for analysis in Coq.

Part II describes the design of the Mathematical Components Analysis library.

Chapter 5 presents my work on the topological hierarchy of the library. It describes how my different contributions fit in this new context.

Chapter 6 explains how we exploited this new hierarchy to develop tools for asymptotic reasoning.

Chapter 7 gives an early assessment of this library based on my experience with it. In particular, I compare the formalisation of the inverted pendulum from Part I with a new one using this library, which may be found in the same repository [CRb].

Part III describes tools for automation that could overcome some of the limitations observed in Chapter 7. This part is focused on proofs by computation.

Chapter 8 introduces a framework that makes it possible to make certified and efficient computations through a separation of concerns.

Chapter 9 explains how we used this framework in the context of proof by reflection. It describes a modular methodology for reflection and its application to a concrete example.

Part I

Case Study: the Inverted Pendulum

Part I
Case Study: the Inverted Pendulum

CHAPTER 1

CONTEXT

This case study deals with the formalisation of the inverted pendulum, which is a standard example in control theory. We give a description of this system and explain why this example is standard in Section 1.1. Then we introduce the mathematics used to model the pendulum in Section 1.2. Finally, we discuss different aspects of the formalisation of such a system and give a clear delimitation to the present work in Section 1.3.

1.1 The Inverted Pendulum

We give a high-level description of the inverted pendulum in Section 1.1.1. Then we briefly discuss in Section 1.1.2 control theory in the perspective of using it to study the inverted pendulum and finally describe the control challenges in the particular case of the pendulum in Section 1.1.3.

1.1.1 The System

As its name indicates, the inverted pendulum is a pendulum, hence a weight which is fixed on one end of a pole. The pole can pivot on its other end. The specificity of this pendulum is that it is inverted: instead of letting gravity bring the weight down, one can **act** on the system to push the weight in the opposite direction.

Indeed, the free [1] pendulum will naturally fall down under the action of gravity and, thanks to friction, eventually stop on a position where the pole is along the vertical line and the weight is on the lower end of the pole (see Figure 1.1a). This state, the position **and** the null velocity, is called an *equilibrium*: once the system reaches this state it cannot leave it. This equilibrium is moreover *stable*, i.e. if the pendulum is slightly perturbed, then it will eventually go back to this state.

1. As in "free of control".

The pendulum has a second equilibrium, still with null velocity and where the pole is along the vertical line, but where the weight is on the upper end of the pole (see Figure 1.1b). This one is *unstable*: after any perturbation, the system will go further and further from this state.



(a) Stable Equilibrium

(b) Unstable Equilibrium

Figure 1.1 – The Two Equilibria of the Free Pendulum

In the case of the inverted pendulum, the unstable equilibrium is the state that matters. However, since the free pendulum will always move away from this state, one has to act on the system to have the weight remain near the upper position. In the way one would balance a ruler on their palm, the mean of action to balance the inverted pendulum is to move the end of the pole on which it pivots. In this thesis, we consider a pendulum where the pivot is on a cart that moves along a one-dimensional horizontal line (see Figure 1.2).

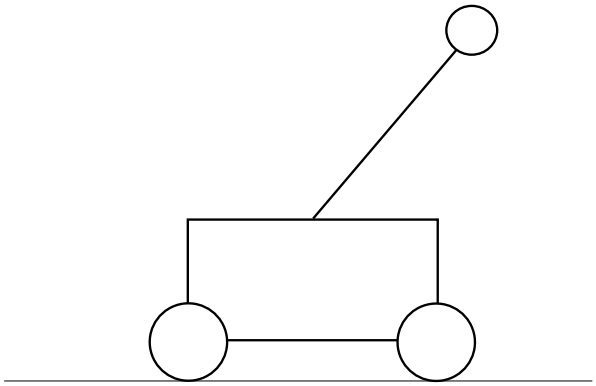


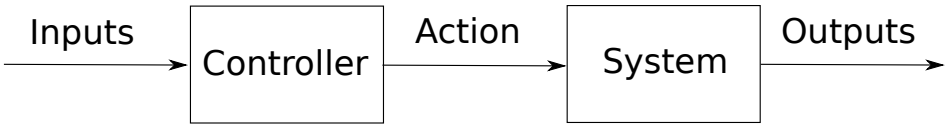
Figure 1.2 – The Inverted Pendulum

By acting on the cart, one can move the pivot and thus affect the rotation of the weight. This action on the cart can be controlled by a computer program: this is where control theory comes into play.

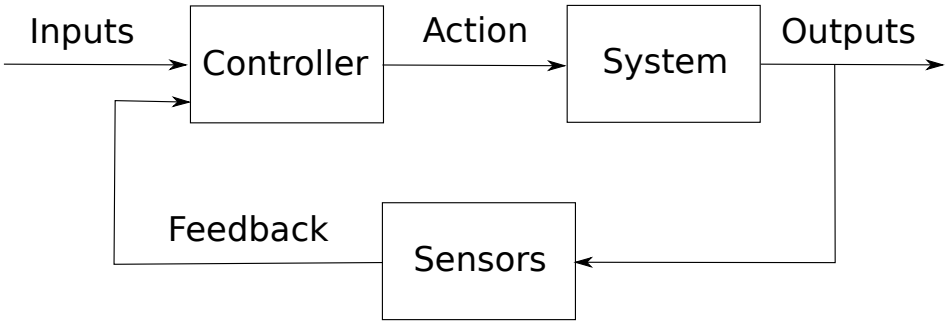
1.1.2 Notions in Control Theory and Application to the Pendulum

Control theory is a field which is concerned with the design of models and tools to grant particular behaviours for dynamical systems. The modelled systems usually operate in continuous time and space, but are often driven by a computer program [ÅM08]. This program, called a controller or control function, takes some inputs and computes an action to apply to the system in order to reach the desired behaviour.

In *open-loop control* (see Figure 1.3a), the controller inputs do not depend on the system outputs. On the contrary, *closed-loop control* (see Figure 1.3b) takes into account a feedback from the system in order to determine the action to apply [MLS94]. This feedback is often obtained from sensors that measure the state of the system [ÅM08] and the problem is to deal with this feedback in an effective way to compute the appropriate response of the controller [LaV06].



(a) Open-Loop Control



(b) Closed-Loop Control

Figure 1.3 – Two Kinds of Control

The inverted pendulum fits in the closed-loop control scheme, as depicted in Figure 1.4: the control function takes as input the state of the system and computes a force to apply to the cart in order to affect the movement of the pivot. The way this force is applied depends on the actual implementation of the pendulum: the choice of the engine will impact the way it is controlled.

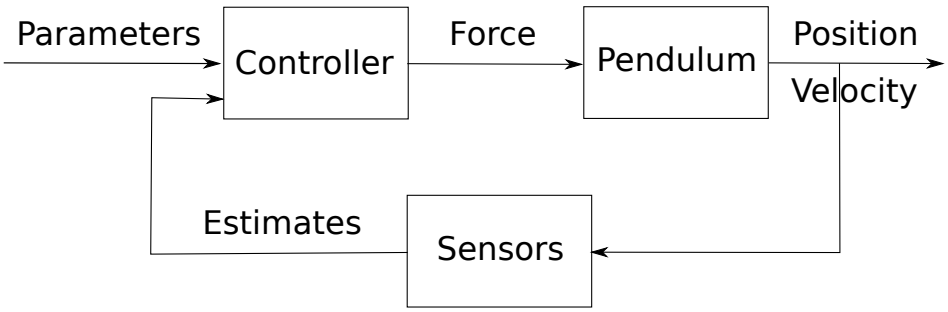


Figure 1.4 – Closed-Loop Control for the Inverted Pendulum

Another point of interest in control theory is the distinction between *linear* and *non-linear* control. Linear control theory deals with dynamical systems which obey to a linear differential equation. General techniques, in particular frequency analysis, apply to linear systems [ÅM08].

Some non-linear systems can be linearised [MLS94], but this is not always the case, so that other (often complex) techniques have been developed [Kha02].

The inverted pendulum fits in the category of non-linear systems, which sometimes can be linearised, depending on the control challenge (see Section 1.1.3 for these challenges). This is thus a simple system that poses not so simple problems, which makes it a good object of study. It is also easy to implement thanks to its simplicity and easy to test thanks to the absence of security constraints (no lives are at stakes). This makes the inverted pendulum a good benchmark for various control techniques, as shown by the luxuriant literature on the topic (Åström and Furuta [ÅF00, introduction] already mention numerous references for the nineties).

1.1.3 Control of the Inverted Pendulum

As explained in Section 1.1.1, the position that actually matters is the unstable equilibrium of the free inverted pendulum. A goal is thus to control the cart in order to reach this equilibrium starting from any position. We say that we perform the *swing-up* of the pendulum. However, perfect physical systems do not exist: there will always be a tiny discrepancy between the theoretical position the system can reach and the position it will actually reach. As a consequence, a second challenge is to stabilise the pendulum in a small region near the equilibrium.

This second challenge makes possible an analogy with the "non-inverted" pendulum: we often teach high-school students to linearise the differential equation representing the movement of the pendulum for small swings, in order to solve it explicitly. By choosing a sufficiently small area near the unstable equilibrium, the equation of the inverted pendulum can also be linearised, thus opening the door to the use of standard techniques in linear control.

In this thesis, we focus on the first challenge: the swing-up of the inverted pendulum. While swing-up is possible without taking into account the environment constraints [ÅF00], a more interesting challenge is to perform the swing-up with a restriction on the space available to the cart [CPJ02], or even with the condition that the cart has to end on its starting position [LFB00].

The inverted pendulum is interesting not only for benchmarking linear and non-linear control techniques [ÅF00], depending on the challenge one considers, but also to model more complex systems. We will cite only two use cases of such models, since they are out of the scope of our case study: object transportation [SMKT96, DSK98] and bipedal motion [SNI02].

1.2 Modelling Physical Systems

The first step in order to reason about a physical system such as the inverted pendulum, either controlled or free, is to build a mathematical model of this system. Such a model is called a dynamical system, which we briefly introduce in Section 1.2.1. We then discuss in Section 1.2.2 how to obtain this dynamical system in practice and how the control function affects such a system, through the example of the inverted pendulum. Finally, we present the notion of *stability*, which generalises the notion of stability of an equilibrium, in Section 1.2.3.

1.2.1 On Dynamical Systems

Physical systems evolve through time. It is thus important to have a mathematical description of this evolution. We only consider deterministic systems here, hence at any given time for any given state there is only one possible evolution. The best case is when one can express the state of the system in function of time. Such a function then expresses all the information about the system and constitutes, together with the state space, what we call a *dynamical system*.

But often times it is only possible to describe the evolution through time of such a function, by determining a differential equation it obeys to, or even, for the worst cases, the evolution through time and space of a quantity that partially describes the state of the system. These worst cases correspond to partial differential equations, illustrated for instance by the equations describing the propagation of a wave in different media [BG94]. In this thesis, we focus on *ordinary* differential equations, hence on the simpler systems for which it is possible to describe the evolution through time of the state.

An ordinary differential equation of order n is described by a function F, a solution being given by a domain D (an interval, included in $\mathbb{R}^+$ in the case of physical systems) and a function y such that

$$\forall t \in D.\ \mathrm{F}\left(t, y(t), y'(t), \ldots, y^{(n)}(t)\right) = 0. \tag{1.1}$$

It is possible to transform an ordinary differential equation of order n into a first-order one by replacing the state $y(t)$ with the vector $z(t) = \left(y(t), y'(t), \ldots, y^{(n-1)}(t)\right)$. Indeed, if (D, y) is a solution of Equation (1.1), then (D, z) is a solution of

$$\forall t \in D.\ G\left(t, z(t), z'(t)\right) = 0, \tag{1.2}$$

where $G(a, b, c) = (c_1 - b_2, c_2 - b_3, \ldots, c_{n-1} - b_n, \mathrm{F}\,(a, b_1, b_2, \ldots, b_n, c_n))$.

Conversely, if we know (D, z) is a solution of Equation (1.2), then we can prove that z is of the form $\left(y, y', \ldots, y^{(n-1)}\right)$, with (D, y) being a solution of Equation (1.1).

An ordinary differential equation of order n is said to be *explicit*, as opposed to the *implicit* description of Equation (1.1), if we can isolate the n^{th} derivative from the others: (D, y) is a solution of the equation [2] if and only if

$$\forall t \in D.\ y^{(n)}(t) = \mathrm{F}\left(t, y(t), y'(t), \ldots, y^{(n-1)}(t)\right). \tag{1.3}$$

It is not always possible to transform an implicit ordinary differential equation into an explicit one [Pet82].

We also say that an ordinary differential equation is *autonomous* if the function F that defines it does not depend on t. We can make an ordinary differential equation autonomous by including time in the state. For instance, if (D, y) is a solution of the first-order ordinary differential equation defined by F, then we define $z(t) = (t, y(t))$ and we prove that (D, z) is a solution of

$$\forall t \in D.\ G\left(z(t), z'(t)\right) = 0, \tag{1.4}$$

2. This function F is not the same as the one in Equation (1.1), but it is the one that is used in practice when discussing such systems.

where $G(a, b) = (b_1 - 1, \mathrm{F}(a_1, a_2, b_2))$.

Conversely, if (D, z) is a solution of Equation (1.4), we can prove that z can be written as $(t + C, y(t))$, where C is a constant and $(D + C, t \mapsto y(t - C))$ is a solution of the first-order equation defined by F, where the domain $D + C$ is the set $\{d + C \mid d \in D\}$.



Remark

We are often interested in ordinary differential equations with initial conditions, i.e. when the values of $y, y', \ldots, y^{(n-1)}$, are fixed for a given time t_0. With such systems, the constant C above can be forced to be 0 by choosing the initial condition $z(t_0) = (t_0, y_0)$.

In Chapter 2, we will focus on first-order explicit autonomous ordinary differential equations, i.e. on equations of the form

$$y' = \mathrm{F} \circ y,$$

where $f \circ g$ is the standard notation for the function composition $t \mapsto f(g(t))$. This form is not however the one we usually obtain when studying physical systems, as we will see in Section 1.2.2.

1.2.2 Taking into Account the Control Function

The dynamical system corresponding to a physical system can be obtained thanks to the laws of physics. While Lagrangian mechanics [Lag11] seems to be the preferred method in robotics, or at least standard enough to constitute a chapter of robotics textbooks [MLS94, CLH+05], it is not the only method to compute the differential equation that describes the system.

Lagrange equations are based on the energy of the system [MLS94], but we get a better intuition of control if we interpret the control function as the source of an external force applied to the free system: control is a perturbation that prevents the system to evolve according to its natural behaviour, in order to obtain a different behaviour. It is thus easier to take into account the effect of the control function if the differential equation is given by a formula based on the forces that apply to the system, such as Newton's second law of motion [New87].

Newton's second law of motion states that the acceleration of a system, i.e. the second derivative of the position, is proportional to the sum of the forces that act on it. Thus, if we are able to list (and compute) all forces that take part in the evolution of the system, then we get a second-order (explicit) differential equation describing this evolution.



Warning

Several notations for differentiation exist [Caj28]. Up to this point, we used Lagrange's notations $\left(y', y'', \ldots, y^{(n)}\right)$, which is quite commonly used in mathematics. For the sake of coherence with textbooks [MLS94, CLH+05, ÅM08], with the paper on the inverted pendulum we formalised [LFB00], and with our own publications [CR17a, Rou18], we will also use Newton's notations $(\dot{y}, \ddot{y}, \ldots)$.

In textbooks [MLS94, CLH+05, ÅM08], the state of the system is usually called a configuration and denoted by q, which is a vector. Thanks to the laws of Newton, it is possible to describe the dynamics of the system by the following standard form:

$$M(q)\ddot{q} + C\left(q, \dot{q}\right)\dot{q} + G(q) + B\left(q, \dot{q}\right) = \tau, \tag{1.5}$$

where $M(q)$ is the *inertia matrix* of the system, $C\left(q, \dot{q}\right)$ represents the *Coriolis forces*, $G(q)$ the *gravitational forces*, $B\left(q, \dot{q}\right)$ the *dissipative forces* such as friction and damping and τ the *external forces*, including the effect of the control function.

Remark

We did not specify in Equation (1.5) on which variables τ depends, since it varies from a system to another and is also function of the kind of control one applies.

In the case of the inverted pendulum, the configuration is given by the position of the cart on the horizontal line, denoted by x, and the angle the pole forms with the vertical line, denoted by θ (see Figure 1.5). Other relevant pieces of information are the length l of the weighted pole, the respective masses m and M of the weight and of the cart (we neglect the pole's mass) and the control force fctrl.

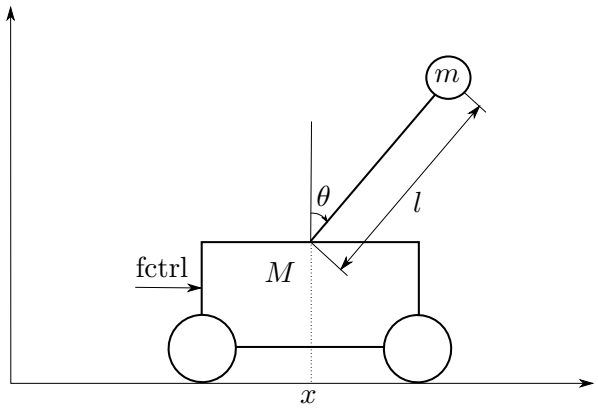


Figure 1.5 – The Inverted Pendulum with Annotations

With these notations, considering the dissipative forces are negligible and using the standard notation g for the gravitational acceleration on Earth, we can rewrite Equation (1.5) for the pendulum as follows [LFB00]:

$$M(q)\ddot{q} + C\left(q, \dot{q}\right)\dot{q} + G\left(q\right) = \tau, \tag{1.6}$$

where

$$q = \begin{pmatrix} x \\ \theta \end{pmatrix}, \qquad C\left(q, \dot{q}\right) = \begin{pmatrix} 0 & -ml\dot{\theta}\sin\theta \\ 0 & 0 \end{pmatrix},$$

$$M(q) = \begin{pmatrix} M+m & ml\cos\theta \\ ml\cos\theta & ml^2 \end{pmatrix}, \qquad G(q) = \begin{pmatrix} 0 \\ -mgl\sin\theta \end{pmatrix},$$

$$\tau = \begin{pmatrix} \text{fctrl} \\ 0 \end{pmatrix}.$$

1.2.3 An Important Property: Stability

As explained in Section 1.1.3, the control challenge of the inverted pendulum is to make it reach a region near its unstable equilibrium where we can stabilise it. In other terms, we want to make the **unstable** equilibrium of the **free** system a **stable** equilibrium of the **controlled** system.

Although the stability of equilibria is the notion that is usually discussed [Kha02], stability is a term that covers several concepts. This notion needs thus to be clarified. In this section I rephrase the presentation of stability by Åström and Murray [ÅM08], which is in my opinion the clearest one. It is focused on dynamical systems that represent physical systems: we consider differential equations with initial conditions at time $t = 0$ and we are interested in the behaviour of solutions for $t > 0$.

Definition 1.1 (Stability)**.** A solution y of an ordinary differential equation with initial condition y_0 is said:

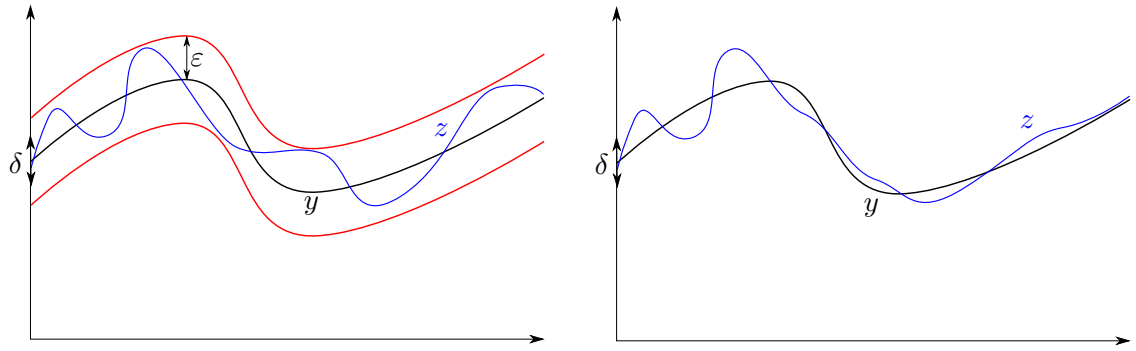
— *stable* (see Figure 1.6a) if and only if $\forall \varepsilon > 0.\ \exists \delta > 0.\ \forall z$ solution with initial condition z_0.

$$\|z_0 - y_0\| < \delta \Rightarrow \forall t > 0.\ \|z(t) - y(t)\| < \varepsilon;$$

— *unstable* if and only if it is not stable;

— *asymptotically stable* (see Figure 1.6b) if and only if it is stable and $\exists \delta > 0.\ \forall z$ solution with initial condition z_0.

$$\|z_0 - y_0\| < \delta \Rightarrow \|z(t) - y(t)\| \xrightarrow[t\to+\infty]{} 0.$$



(a) Stable solution (b) Asymptotically Stable Solution

Figure 1.6 – Illustration of the Notions of Stability

This notion of stability is called *Lyapunov stability*, in reference to Aleksandr Lyapunov's work [Lia07].



Remark

Lyapunov stability may be used to define the notion of stable equilibrium. Indeed, no solution can leave an equilibrium point, hence to an equilibrium point p corresponds a solution which is a constant function: the function $t \mapsto p$. We say that an equilibrium is (asymptotically) stable if the corresponding solution is (asymptotically) stable.

The intuition we gave in Section 1.1.1 on stable equilibria thus corresponds to asymptotically stable equilibria. Although stability is sufficient in some cases, for instance to keep the pendulum in a small region around the equilibrium, asymptotic stability is a more desirable property.

We also sometimes abusively say that a dynamical system is (asymptotically) stable when its equilibrium point is (asymptotically) stable.

LaSalle [LaS60, LaS76] proposes another point of view on stability in order to extend Lyapunov's ideas, based on the notion of stable set. We will discuss this notion in Section 2.1.2.

1.3 Practical Aspects of such a Study

In order to formally study a physical system, one has to take many factors into account. We enumerate these factors and suggest a possible strategy for considering them separately in Section 1.3.1. We then give the exact scope and goal of our case study in Section 1.3.2.

1.3.1 How to Formally Study a Physical System

As explained in Section 1.2, the first step to study a physical system is to build a mathematical model. However, differences between the physical system and its model are bound to occur. Already during this first step we have to make approximations: in order to obtain Equation (1.6) in Section 1.2.2, we had to neglect the mass of the pole as well as friction. We also assumed the cart was moving on a straight horizontal line, in an empty environment (no obstacle, no possible perturbation), which is obviously not the case in the real world: the cart wheels are not exactly parallel, the ground is not perfectly flat and horizontal and obstacles and perturbations can occur (a meteorite could perfectly fall on the pendulum...).

These differences are inherent to modelling: a model is a simplified version of reality that allows us to focus on the relevant pieces of information. Knowing how far one can go in the formal description of this simplification in order to reach a high level of confidence in the physical system is a tough question. The more parameters one takes into account, the more complex the model will be and the harder it will be to prove properties on this model. We must however take into account sufficiently many parameters to reach the appropriate level of confidence.

In my opinion, two kinds of difference must be considered. On one hand, systems such as the inverted pendulum evolve in a continuous world but are driven by a computer program. This program is executed on a processor which has a given frequency: it operates in discrete time. This means that the control function that is computed does not vary continuously but is a piecewise constant function (with an arbitrary number of pieces). Thus, the system necessarily deviates from the ideal solution of the differential equation where the feedback is continuous.

On the other hand, one also has to take into account approximations, which are of several kinds. First, the fact that the control function is piecewise constant is in itself an approximation. The computed control force must be applied through the action of the engine, which cannot instantly reach any force: there is a "latency", the discontinuities of the discrete model are in fact replaced with continuous pieces with a high slope (e.g. as in Figure 1.7).

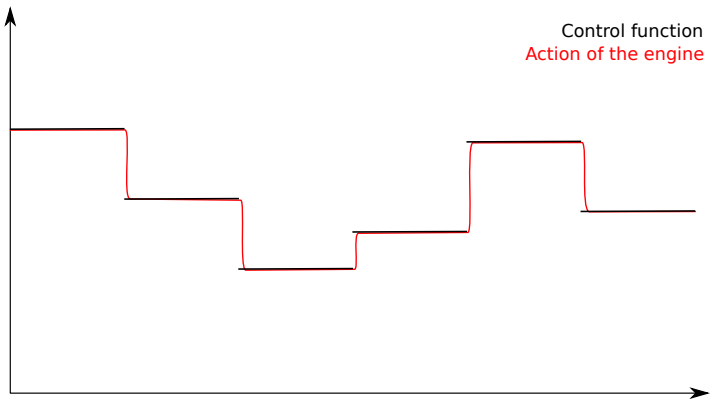


Figure 1.7 – The Computed Control Function

Then, the control function is not "just" a mathematical function given by a formula, it is computed by a program. The program that runs on the system's processor does not compute using exact real numbers but performs numerical computations using floating-point numbers, which means there are rounding errors. Finally, another kind of approximation is introduced by the sensors: they do not measure the exact state of the system but evaluate it with a given precision.

This classification in discretisation and approximation errors generalises the "*method*" and "*round-off*" *errors* for numerical approximation schemes (using the same terminology as Boldo et al. [BCF+13]). The method error, sometimes also called *truncation error*, is the error one makes by using an approximation scheme to estimate a solution of a differential equation. Discretisation is analogous to the use of such a scheme. Although round-off error corresponds only to errors in computation, I believe the approximations related to sensor precision and to the engine response to discontinuities could be taken into account through an additional incertitude on the state of the system. Moreover, since processors have a high frequency, if the control function is smooth enough the discontinuities should be small and the engine latency negligible.

Once these differences between the model and the concrete system are taken into account, one also has to verify that the program is executed in a proper way. In particular, unless one wants to program using the instruction set of the processor of the system, one must prove that compilation does not introduce errors, for instance using a certified compiler [KLB+17, ABB+17].

To sum up, we can establish the following strategy for the formal study of a system such as the inverted pendulum, illustrated by Figure 1.8. First, select the relevant characteristics of the system and discard the ones considered to be negligible. The control function is one of the relevant characteristics. Then, build a mathematical model according to the laws of physics. This model usually helps to design the formula which defines the control function and thus

the program that has to be implemented. This makes it possible to prove properties on the solutions of the obtained differential equation. Then, one has to take into account the impact of discretisation and of approximations and to verify that these properties remain true. At this point, a high-level implementation of the control function is proven correct. The last step is to prove that the compiled version of the program keeps these properties.

1.3.2 Scope of our Case Study

The strategy illustrated by Figure 1.8 is actually a vast programme, which involves different formalisation techniques. The last step means proving the correspondence between two implementations of a same program in two different programming languages. Taking into account approximations implies proving that the same algorithm keeps its properties while using different data structures (we switch from exact computations on real numbers to approximate computations on floating-point numbers). This is all different from studying the behaviour of the solutions of a differential equation.

Having a predilection for pure mathematics, I focus in this thesis on the analysis step: proving properties of the solutions of a given differential equation, namely Equation (1.6). This means that I assume the differential equation is correct, i.e. it accurately models the system, and I only consider pure mathematical functions dealing with exact real numbers.

This step can be tackled in two ways: either by a *qualitative analysis* of solutions, or by a *quantitative analysis*, i.e. either by considering abstract solutions and deriving properties from the fact that they obey to the differential equation, or by resolving the equation and then studying the properties of the concrete solutions. Resolving here means finding an analytical solution, but one could consider a more general definition of quantitative analysis where numerical resolution is allowed as long as it is possible to bring guarantees on the accuracy of the approximation.

This case study is in fact a formalisation of a proof by Lozano et al. [LFB00], who perform a qualitative analysis of the solutions of Equation (1.6). They prove that the control function they give makes the solutions of Equation (1.6) achieve the swing-up of the inverted pendulum, with the additional condition that the cart stops on its initial position.

Before discussing their proof and our formalisation thereof (see Chapter 3), we focus in Chapter 2 on a standard theorem in stability analysis which is at the core of this proof: LaSalle's invariance principle [LaS60, LaS76].

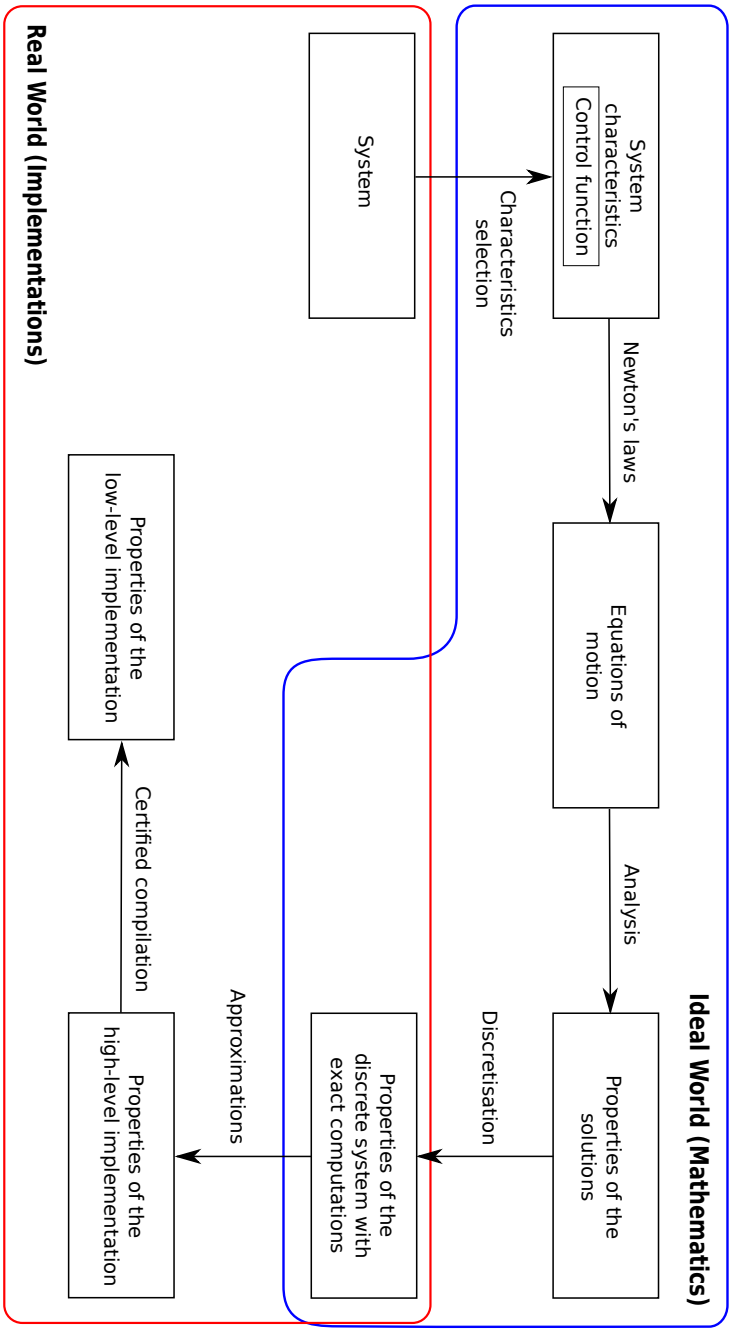


Figure 1.8 – Steps in the Formal Study of a Physical System

Part I

Case Study: the Inverted Pendulum

CHAPTER 2

LASALLE'S INVARIANCE PRINCIPLE

LaSalle's invariance principle [LaS60, LaS76] states a sufficient condition for the asymptotic stability of the solutions of a first-order explicit autonomous ordinary differential equation. Asymptotic stability being a crucial notion in the study of non-linear systems [Kha02], several versions of this principle exist [LaS60, LaS76, CLH99, CZČ05, MG06, AE09, FKD13, Bar14, Bar17]. Our case study is based on the original invariance principle [LaS60], which we explain in Section 2.1. While working on its proof, we noticed that straightforward modifications could be done to generalise this theorem, leading to yet another version described in Section 2.2. Then we give more technical details on the formalisation in Section 2.3 and finally discuss related work in Section 2.4.

This chapter may be seen as an extended version of our publication on the topic [CR17a], updated with later work that improves the version of the theorem that is formalised [Rou18]. This work was done in collaboration with Cyril Cohen, who mainly contributed to the notations and to the inference mechanism presented in Section 2.3.2. We give here the first detailed description of this inference mechanism. All code snippets come from our on-line repository [CRa], unless otherwise specified.

2.1 The Original Principle

LaSalle published several versions of the invariance principle [LaS60, LaS76], but the original one seems to be the one that is most commonly used [MLS94, Kha02, ÅM08]. We first discuss in Section 2.1.1 the intuition behind this theorem and then give its mathematical statement in Section 2.1.2. Finally, we give its proof in Section 2.1.3.

2.1.1 Intuition behind the Invariance Principle

LaSalle's invariance principle [LaS60] is an extension of Lyapunov's work about stability [Lia07]. The idea of Lyapunov is that if one can find a function satisfying a few conditions with respect to the differential system, then one can guarantee stability. LaSalle's work in fact weakens the conditions on this Lyapunov function. We will now discuss these conditions.

We consider an equation of the following form.

$$y' = \mathrm{F} \circ y. \tag{2.1}$$

We thus have a function F which defines the velocity of solutions in function of their position. This function can be seen as a vector field: at each point of the space corresponds a vector which represents the velocity of a solution when it goes through this point. Visualising this vector field can give clues on how solutions are expected to behave (e.g. see Figure 2.1).

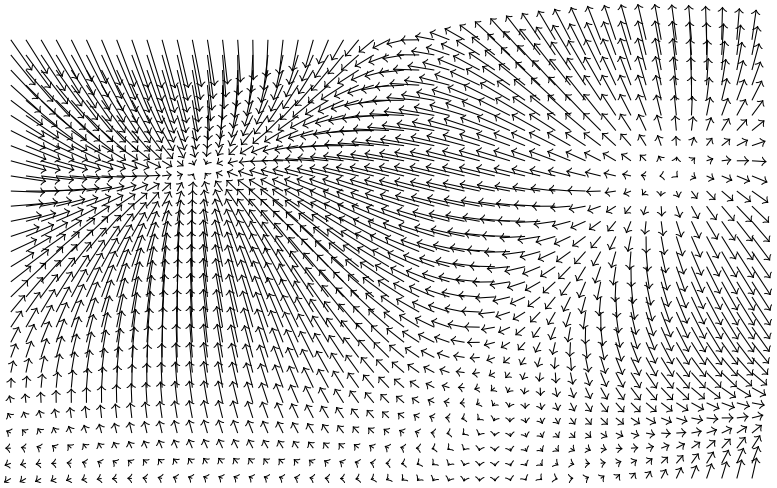


Figure 2.1 – A Vector Field

The main condition on Lyapunov functions is that they must be akin to potential functions: a Lyapunov function V is a scalar function that decreases along the trajectories of solutions of Equation (2.1). A more illustrative way of putting this is that the vector field defined by F must drive solutions downwards on the contour map of V (see Figure 2.2 for an example corresponding to Figure 2.1).

If the Lyapunov function is well-chosen, the trajectories of solutions will converge to a plateau of V and the solutions will not be able to leave it (this plateau is quite visible in our example of Figure 2.2). This is the essence of LaSalle's invariance principle.

Now, the difficulty is to find a Lyapunov function which is appropriate enough for this plateau to grant the sought properties. Sometimes, one function is not enough and one has to combine the properties of several Lyapunov functions to obtain the desired result [Mat62, SPA16]. In the case of the paper on the inverted pendulum we formalised [LFB00], one Lyapunov function was sufficient to prove the stability of the system.

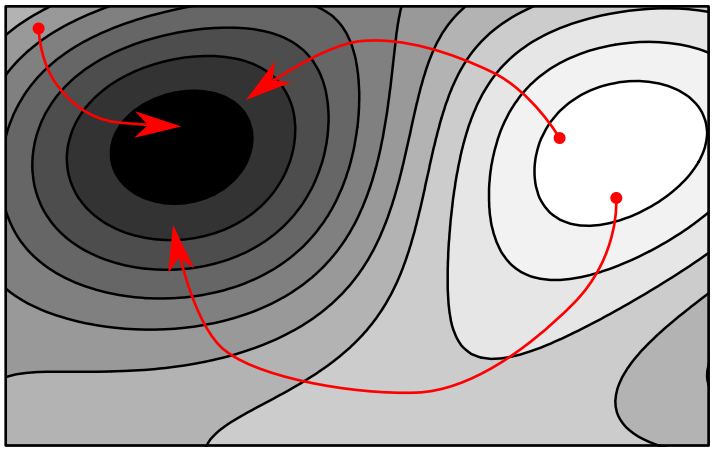


Figure 2.2 – Contour Map of a Lyapunov Function

2.1.2 Statement of LaSalle's Invariance Principle

In order to precisely state the original version of LaSalle's invariance principle [LaS60], we first have to give the mathematical definition of the different ingredients described in Section 2.1.1. We consider an ordinary differential equation in $\mathbb{R}^n$, i.e. the function F has type $\mathbb{R}^n \to \mathbb{R}^n$. A Lyapunov function is a scalar function, i.e. of type $\mathbb{R}^n \to \mathbb{R}$. The condition that a Lyapunov function V must decrease along the trajectories of solutions of Equation (2.1) is expressed as the condition $\tilde{V} \leqslant 0$, where $\tilde{V}$ is defined as follows.

Definition 2.1. Let V be a scalar function with continuous first partial derivatives. Define, for all $p \in \mathbb{R}^n$,

$$\tilde{V}(p) = \langle (\operatorname{grad} V)(p), F(p) \rangle ,$$

where $\langle ., . \rangle$ is the scalar product of $\mathbb{R}^n$ and $\operatorname{grad} V$ is the gradient of V.

Remark

An important property of $\tilde{V}$ is that if y is a solution of Equation (2.1), then the derivative of $V \circ y$ is $\tilde{V} \circ y$. Indeed, for all p,

$$\tilde{V}(p) = \langle (\operatorname{grad} V)(p), F(p) \rangle = (\mathrm{d}\, V_p \circ F)(p),$$

where $\mathrm{d}\, V_p$ denotes the differential of V at point p. As a consequence, the chain rule of differentiation gives, for all $t \geqslant 0$,

$$(\tilde{V} \circ y)(t) = (\mathrm{d}\, V_{y(t)} \circ F \circ y)(t) = (\mathrm{d}\, V_{y(t)} \circ \dot{y})(t) = (V \circ y)'(t).$$

Thus, if $\tilde{V} \circ y$ is always non-positive, then V is indeed non-increasing along the trajectory of y.

If $\tilde{V}$ satisfies this sign condition, then the solutions of Equation (2.1) must converge to a plateau of V, i.e. a set where $\tilde{V}$ is constantly null. More precisely, this plateau is proven to be asymptotically stable: LaSalle generalises the notion of stability (recall Definition 1.1) to sets (see Definition 2.3). The idea is to replace the distance between points with the distance of a point to a set (see Definition 2.2).

Definition 2.2 (Convergence to a set)**.** The distance from a point p to a set E is the minimum distance from p to a point $q \in E$.

$$\mathrm{d}(p, E) = \min_{q \in E} \|p - q\| .$$

We say that the function y converges to a set E as time goes to infinity, if

$$\forall \varepsilon > 0.\ \exists T > 0.\ \forall t > T.\ \mathrm{d}\left(y(t), E\right) < \varepsilon.$$

Remark

Since convergence to a set is a straightforward generalisation of convergence to a point, we use the same notation to denote convergence: $y(t) \xrightarrow[t \to +\infty]{} E$.

See Figure 2.3 for an illustration of Definition 2.2.

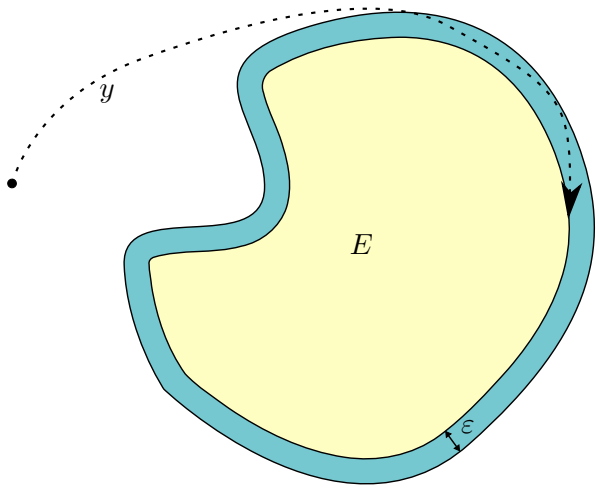


Figure 2.3 – Illustration of Convergence to a Set

Definition 2.3 (Stable sets)**.** A set E is:

— stable if and only if $\forall \varepsilon > 0.\ \exists \delta > 0.\ \forall y$ solution with initial condition y_0.

$$\mathrm{d}\left(y_0, E\right) < \delta \Rightarrow \forall t > 0.\ \mathrm{d}\left(y(t), E\right) < \varepsilon;$$

— asymptotically stable if and only if it is stable and $\exists \delta > 0.\ \forall y$ solution with initial condition y_0.

$$\mathrm{d}\left(y_0, E\right) < \delta \Rightarrow y(t) \xrightarrow[t \to +\infty]{} E.$$

Remark

Definition 2.3 gives another equivalent definition of the stability of an equilibrium point: a stable equilibrium is a stable *invariant* singleton set (invariance is introduced in Definition 2.4).

Note that the plateau of V to which the solutions converge is also invariant.

Definition 2.4 (Invariant Sets)**.** The set E is said to be invariant if every solution y of Equation (2.1) starting in E, i.e. $y(0) \in E$, remains in E, i.e. $\forall t \geqslant 0.\ y(t) \in E$.

The notion of invariant set is quite important. Usually, stability cannot be obtained globally. For instance, the stable equilibrium of the free pendulum is only locally stable: not any starting position makes the pendulum converge to this point, since starting on the unstable equilibrium gives a trajectory that will not converge to the stable equilibrium. Finding the equilibrium's *basin of attraction*, i.e. the set of starting positions such that the system will indeed converge to this point, can be hard. Finding an invariant set that contains the equilibrium is easier and sufficient in LaSalle's invariance principle (as long as one also finds a Lyapunov function).

We can now state LaSalle's principle [LaS60], illustrated by Figure 2.4.

Theorem 2.1 (LaSalle's Invariance Principle)**.** *Assume* F *has continuous first partial derivatives and* $\mathrm{F}(0) = 0$*. Let* K *be an invariant compact set. Suppose there is a scalar function* V *which has continuous first partial derivatives in* K *and is such that* $\tilde{V}(p) \leqslant 0$ *in* K*. Let* E *be the set of all points* $p \in K$ *such that* $\tilde{V}(p) = 0$*. Let* M *be the largest invariant set in* E*.*

Then for every solution y *of Equation* (2.1) *starting in* K*,* $y(t) \xrightarrow[t \to +\infty]{} M$*.*

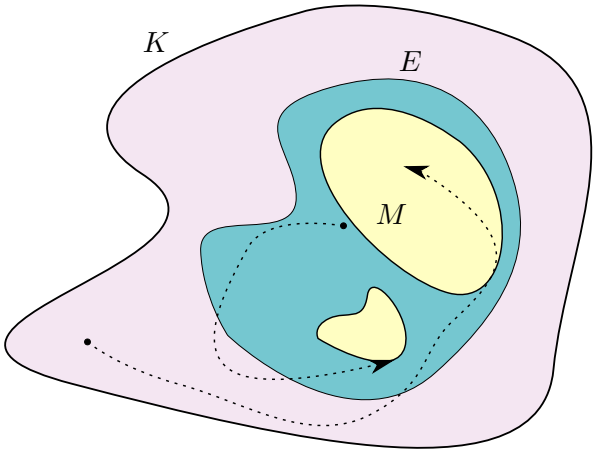


Figure 2.4 – Illustration of the Original Invariance Principle

2.1.3 Proof of the Invariance Principle

Let us briefly explain LaSalle's proof of Theorem 2.1 [LaS60]. Under the assumptions of Theorem 2.1, the goal is to prove that any solution of Equation (2.1) starting in K converges to M. So, let y be a solution of Equation (2.1) starting in K.

An important notion to study y's asymptotic behaviour is its set of limit points (see Definition 2.5), i.e. the set of points that the trajectory of y approaches arbitrarily close an infinite number of times as time goes to infinity.

Definition 2.5 (Positive Limiting Set)**.** Let y be a function of time. The positive limiting set of y, denoted by $\Gamma^+(y)$, is the set of all points p such that

$$\forall \varepsilon > 0.\ \forall T > 0.\ \exists t > T.\ \|y(t) - p\| < \varepsilon.$$

The most interesting property of positive limiting sets is that, even though y may not have a limit, y converges to its set of limit points (see Lemma 2.2). This means in particular that to prove that y converges to M, it is sufficient to prove that $\Gamma^+(y) \subseteq M$. We do this by proving that $\Gamma^+(y)$ is an invariant subset of E (see Lemma 2.4 and Lemma 2.5), which concludes since M is the largest such set. This proof relies on properties of solutions that are stated in Lemma 2.3.

Lemma 2.2. *Let y be a function of time with values in a compact set.*
Then $y(t) \xrightarrow[t\to+\infty]{} \Gamma^+(y)$.

Proof. This proof goes by contradiction. Assume y does not converge to its positive limiting set. Then for some $\varepsilon > 0$, and all $T > 0$, there exists $t > T$ such that $d\left(y(t), \Gamma^+(y)\right) \geqslant \varepsilon$.

Thus, we can build a sequence $(t_n)_{n\in\mathbb{N}}$ such that $t_n \xrightarrow[n\to\infty]{} +\infty$ and for all $n \in \mathbb{N}$, $d\left(y\left(t_n\right), \Gamma^+(y)\right) \geqslant \varepsilon$.

However, since all $y\left(t_n\right)$ are in a compact set, we know the sequence $\left(y\left(t_n\right)\right)_{n\in N}$ has a limit point. This limit point is in $\Gamma^+(y)$ (use t_n for t with an appropriate n in Definition 2.5). But then, we know $d\left(y\left(t_n\right), \Gamma^+(y)\right) < \varepsilon$ for n large enough, which is a contradiction. □

Lemma 2.3. *We have the existence and uniqueness (given an initial value) of solutions of Equation (2.1).*

Moreover, we have the continuity of solutions of Equation (2.1) with respect to initial conditions, i.e. for all solution y starting at y_0, all time $t \geqslant 0$ and all $\varepsilon > 0$, there exists $\delta > 0$ such that for all solution z starting at z_0 such that $\|y_0 - z_0\| < \delta$, we have $\|y(t) - z(t)\| < \varepsilon$.

Proof. Since F has continuous first partial derivatives, F is of class C^1, hence it is Lipschitz continuous and the Cauchy-Lipschitz Theorem (also known as the Picard-Lindelöf Theorem) concludes. □

Lemma 2.4. *$\Gamma^+(y)$ is invariant.*

Proof. Let z be a solution of Equation (2.1) starting in $\Gamma^+(y)$. We want to prove that for all $t \geqslant 0$, $z(t) \in \Gamma^+(y)$. So, let $t_0 \geqslant 0$, $\varepsilon > 0$ and $T > 0$. The goal is to find some $t > T$ such that $\|z\left(t_0\right) - y(t)\| < \varepsilon$.

By the continuity of solutions of Equation (2.1) with respect to initial conditions (see Lemma 2.3), we know that there exists $\delta > 0$ such that for all solution f of Equation (2.1) such that $\|z(0) - f(0)\| < \delta$, we have $\|z\left(t_0\right) - f\left(t_0\right)\| < \varepsilon$. We also know that $z(0) \in \Gamma^+(y)$, hence there exists $t_1 > T$ such that $\|z(0) - y\left(t_1\right)\| < \delta$.

Thanks to the existence of solutions of Equation (2.1) (see Lemma 2.3), there is a solution f such that $f(0) = y\left(t_1\right)$. Thus, we know that $\|z\left(t_0\right) - f\left(t_0\right)\| < \varepsilon$.

Using again Lemma 2.3 for the uniqueness of solutions of Equation (2.1), we have for all $t \geqslant 0$, $f(t) = y(t_1 + t)$. In particular, $f(t_0) = y(t_1 + t_0)$, hence $\|z(t_0) - y(t_1 + t_0)\| < \varepsilon$. Since we know $t_1 + t_0 > T$, this concludes the proof. □

Lemma 2.5. $\Gamma^+(y) \subseteq E$.

Proof. Since K is invariant and y starts in K, we know that for all $t \geqslant 0$, $y(t) \in K$. Hence, for all $t \geqslant 0$, we have $\tilde{V}(y(t)) \leqslant 0$. Thus, $V \circ y$ is non-increasing (recall the remark following Definition 2.1).

Moreover, V is continuous on K, since it has continuous first partial derivatives in K. Thus, since y takes values in the compact set K, and since the continuous image of a compact set is compact, we know that $V \circ y$ is bounded. Since $V \circ y$ is non-increasing and bounded, it converges to a finite limit l as time goes to infinity.

Since y takes values in K, which is closed, its limit points all belong to K, i.e. $\Gamma^+(y) \subseteq K$. Using the continuity of V on K and the fact that $(V \circ y)(t) \xrightarrow[t \to +\infty]{} l$, we can thus prove that $V(\Gamma^+(y)) = \{l\}$.

Let then $p \in \Gamma^+(y)$ and z be the unique solution of Equation (2.1) starting at p (see Lemma 2.3). By Lemma 2.4 and what precedes, we know that for all $t \geqslant 0$, $V(z(t)) = l$. Thus, by differentiation, $\tilde{V} \circ z$ is constantly null. In particular at time 0, $\tilde{V}(p) = 0$, hence $p \in E$. □

2.2 Generalisation of LaSalle's Invariance Principle

Observing the proof in Section 2.1.3, we can already notice a few elements that can lead to a straightforward generalisation of LaSalle's invariance principle. In particular, we can relax some hypotheses (see Section 2.2.1) and have a stronger conclusion (see Section 2.2.2). This was in fact intended by LaSalle who first gave an illustration of his ideas [LaS60] and later proposed another version of the invariance principle with weaker assumptions [LaS76]. Still, we focused on the original version which is still the one that is explained in modern textbooks [MLS94, Kha02, ÅM08].

2.2.1 Weaker Hypotheses for the Invariance Principle

In his original paper [LaS60], LaSalle wants to illustrate his method and hence makes any assumption that makes it easier to present. In particular, LaSalle wants to discuss the stability of equilibria. Hence, he not only assumes there is an equilibrium, but also that this equilibrium is conveniently located at the origin ($\mathrm{F}(0) = 0$). This hypothesis does not play any role in the proof we gave in Section 2.1.3. It is only important in LaSalle's examples where there is indeed an equilibrium at the origin and V is such that $M = \{0\}$. As a consequence, we removed this hypothesis.

The other hypothesis on the vector field F (it has continuous first partial derivatives) is also a convenience. LaSalle in fact needs "any other conditions that guarantee the existence and uniqueness of solutions and the continuity of the solutions relative to the initial conditions". In his later work [LaS76], LaSalle only assumes that these three properties hold on a subset of the ambient space, and that F is continuous on this subset. Assuming these properties on a subset of the ambient space is important since it may happen that the vector field is not defined everywhere (e.g. if the system comes from a control function that has singularities,

see for instance Section 3.1). However, the continuity of F is not required as long as we still have the existence and uniqueness of solutions of Equation (2.1) in K and the continuity of solutions of Equation (2.1) with respect to initial conditions.

The regularity assumption on V is also too strong. What we actually used in the proof of Lemma 2.5 is the continuity of V and the fact that $\tilde{V} \circ y$ is the derivative of $V \circ y$ when y is a solution of Equation (2.1). In our first published work on LaSalle's invariance principle [CR17a], we assumed that V is differentiable in K. We later realised that this was still too strong and that we could assume that V is only continuous on K and that for all y solution of Equation (2.1), $V \circ y$ is derivable at any time.

This last assumption makes Definition 2.1 unusable since the gradient of V is not necessarily well-defined. We can however use the remark following this definition to give an alternative formulation of $\tilde{V}$ (see Definition 2.6).

Definition 2.6. Let V be a scalar function which is differentiable along the trajectories of solutions of Equation (2.1), i.e. $V \circ y$ is derivable at any time for all solution y of Equation (2.1). Define, for all $p \in \mathbb{R}^n$,

$$\tilde{V}(p) = (V \circ y)'(0),$$

where y is the unique solution of Equation (2.1) starting at p.



Warning

Since the existence and uniqueness of solutions of Equation (2.1) are required in Definition 2.6, $\tilde{V}$ is only defined in the compact set K, where we assume these properties.

We still have to check that this definition indeed provides us with the desired property, which is the purpose of Lemma 2.6.

Lemma 2.6. *For any solution y of Equation (2.1) starting in K, $\tilde{V} \circ y$ is the derivative of $V \circ y$.*

Proof. Let $t \geqslant 0$. We want to prove that $\tilde{V}(y(t)) = (V \circ y)'(t)$.

By assumption, $V \circ y$ is derivable at time t. By Definition 2.6, $\tilde{V}(y(t)) = (V \circ z)'(0)$, where z is the unique solution of Equation (2.1) starting at $y(t)$.

By the uniqueness of solutions of Equation (2.1), we know that for all $s \geqslant 0$, $z(s) = y(t+s)$, hence for all $s \geqslant 0$, $(V \circ z)(s) = (V \circ y)(t+s)$. By differentiation at time 0, we know that

$$(V \circ y)'(t) = (V \circ z)'(0) = \tilde{V}(y(t)).$$

□



Remark

Lemma 2.6 gives us another intuition on LaSalle's invariance principle: the condition $\tilde{V}(p) = 0$ means that the Lyapunov function V converges to a minimum, since it is decreasing along the trajectories of solutions according to Lemma 2.6 and to the sign condition on $\tilde{V}$.

Using Definition 2.6 instead of Definition 2.1 makes it possible to relax a last hypothesis. Indeed, since we do not use the gradient any more, the ambient space does not need to be $\mathbb{R}^n$, nor does it need to be a finite-dimensional vector space. In fact, we managed to prove LaSalle's invariance principle in any normed module over $\mathbb{R}$.

2.2.2 A Stronger Invariance Principle

LaSalle's proof [LaS60] of Theorem 2.1, which we described in Section 2.1.3, shows the convergence of the solutions of Equation (2.1) to M by using the properties of positive limiting sets. In fact, the maximality of M plays a minor role. What is truly important for applications of LaSalle's invariance principle is the fact that M is an invariant subset of E. Combining invariance with the fact that $\tilde{V}$ is constantly null on M already gives all desirable properties. For instance, in LaSalle's paper [LaS60], M is always reduced to the equilibrium of the system.

Since for any solution of Equation (2.1) starting in K, we know by Lemmas 2.4 and 2.5 that its positive limiting set is already an invariant subset of E, all desirable properties already hold on this smaller set. We can then strengthen the conclusion of LaSalle's invariance principle by replacing M with the positive limiting set of the considered solution (see Theorem 2.7, illustrated by Figure 2.5).

Theorem 2.7 (A Stronger Invariance Principle)**.** *Assume F is such that we have the existence and uniqueness of solutions of Equation (2.1) in an invariant compact set K and the continuity of solutions with respect to initial conditions in K. Suppose there is a scalar function V, continuous on K, such that for all solution y of Equation (2.1) starting in K, $V \circ y$ is derivable at any time, and that for all point $p \in K$, $\tilde{V}(p) \leqslant 0$. Let E be the set of all points $p \in K$ such that $\tilde{V}(p) = 0$.*

Then, for all solution y of Equation (2.1) starting in K, $\Gamma^+(y)$ is an invariant subset of E and $y(t) \xrightarrow[t \to +\infty]{} \Gamma^+(y)$.

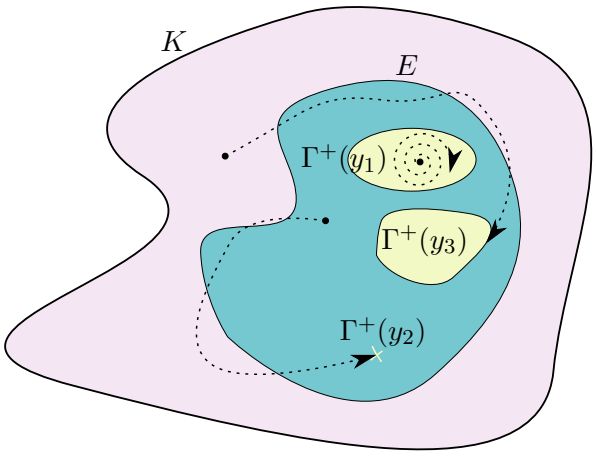


Figure 2.5 – Illustration of the Stronger Invariance Principle

Remark

We also proved a slightly weaker version of Theorem 2.7, where we replace $\Gamma^+(y)$ with the union of all $\Gamma^+(y)$ for y solution of Equation (2.1) starting in K. This defines an asymptotically stable set which does not depend on the solution y.

2.3 Formalisation of the Generalised Principle

We formalised Theorem 2.7 in the Coq proof assistant using the SSReflect tactic language [GMT15]. For real analysis, we chose to use the Coquelicot library [BLM15], which is an extension of Coq's standard library for real analysis [May01], amongst other libraries. We explain our choice in Section 2.3.1. We then discuss the formalisation of the two main notions involved in Theorem 2.7: convergence (see Section 2.3.2) and compact sets (see Section 2.3.3). Finally, we give the formal statement of Theorem 2.7 in Section 2.3.4.

2.3.1 A Note on Logical Foundations and Choosing a Library

Several libraries for real analysis exist in Coq. Let us say a few words about those we know of, before explaining our choice of the Coquelicot [BLM15] library.

Warning

It is important to have a basic knowledge of logic for this section to make sense.

Coq's logic is *constructive*, i.e. one has to provide an explicit witness in order to prove the existence of a particular object. In particular, this implies that the law of Excluded Middle (see EM below) is not provable using Coq. In order to use it, one has to pose it as an additional axiom using the `Axiom` command.

$$\frac{}{\Gamma \vdash P \vee \neg P}\ \text{EM}$$

We sometimes use the term *intuitionist logic* to denote constructive logic. We say that the logic is *classical* when its axioms entail EM.

Thanks to Coq's constructive logic, the decidability of a predicate `P : T -> Prop` corresponds to the existence of a function `f : T -> bool` that is logically equivalent to the predicate. Indeed, thanks to the computational content of `bool`, such a function is an algorithm that decides in effect if the proposition holds on a given argument. Since witnesses are explicit in intuitionist logic, proving the existence of such a function amounts to implementing the corresponding algorithm. One may state such an equivalence as a *reflection view* (using the `reflect` inductive), which is a common practice in Mathematical Components [MT18]:

```
forall t : T, reflect (f t) (P t).
```

We may also use Coq's *constructive sum*, `{P} + {Q}`, to denote the possibility of deciding between two propositions `P` and `Q`. The constructive sum plays the role of the

disjunction `P ∨ Q` but has a computational content that can be used to define functions with values in a given data set depending on whether `P` or `Q` is true.

With strong enough axioms (such as the ones we will present in Section 5.1.1), the existence of a boolean predicate does not mean decidability any longer, since it is possible to prove such an existence even for undecidable predicates. We will say by abuse of language that these predicates become decidable: we call decidable any predicate `P` for which there exists a function `f` computing boolean values and that is logically equivalent to `P`.

Coq's standard library

Coq's standard library contains a part for real analysis that was first developed by Mayero [May01]. It is based on a **classical axiomatisation** of the set of real numbers: this set is represented by an abstract type `R` together with axioms that make it a totally ordered archimedean field equipped with an upper bound function. What makes this axiomatisation classical is the characterisation of the total order:

```
Axiom total_order_T : forall r1 r2 : R, {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

Axiom `total_order_T` expresses the fact that both equality and strict comparison are decidable on real numbers. This axiom makes it possible to develop a theory of real analysis without adding EM to Coq's logic.

This library contains theories about limits, derivatives, series, elementary functions and Riemann integrals.

The Coquelicot Library

Coquelicot [BLM15] is a *conservative extension* of Coq's standard library for real analysis, i.e. no new axiom is added, what is more compatible with the standard library: Coquelicot redefines in a provably equivalent way some notions such as limits and derivatives.

Coquelicot makes use of the theories of general algebra and general topology in order to derive a general theory that is not limited to real analysis. To this end, Coquelicot contains a hierarchy of algebraic and topological structures, depicted in Figure 2.6.

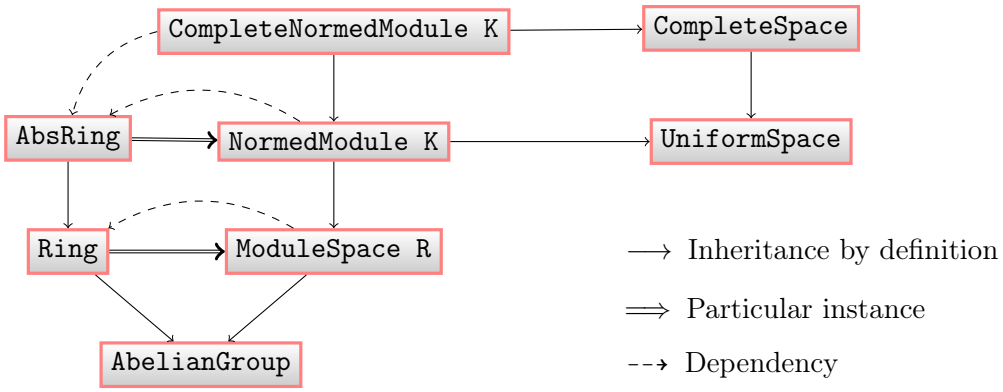


Figure 2.6 – The Coquelicot Hierarchy

The algebraic structures in Coquelicot are abelian groups (`AbelianGroup`), rings (`Ring`) and modules over a given ring (`ModuleSpace R`). The most basic topological structure in Coquelicot is the uniform space (`UniformSpace`), which is a particular kind of topological space. Such a space can be complete (`CompleteSpace`). When a ring is equipped with an absolute value (`AbsRing`), modules over such a ring that are equipped with a norm (`NormedModule K`) are uniform spaces. These spaces can be complete too (`CompleteNormedModule K`). We will more precisely discuss this hierarchy in Section 5.2.

A great part of the library deals with uniform spaces and normed modules. Some parts specifically concern the set of real numbers, which forms a complete normed module over itself thanks to the axioms of the standard library.

The CoRN and MathClasses Libraries

The CoRN [CGW04] and MathClasses [SvdW11, KS11] libraries form a **constructive** library for real analysis. The library is still built from an abstract interface with a carrier type and axioms it has to satisfy. However, this interface is bundled in a record one can instantiate, which has been done by defining real numbers as rational Cauchy sequences [GN00].

An important aspect of constructive real analysis is that equality is no longer decidable. This rules out some commonly-used proof techniques and makes it necessary to reformulate some notions, which has a great impact on the formalisation. Mayero details this incompatibility between Coq's standard library and CoRN in her habilitation [May12].

The COQTAIL Library

The COQTAIL library [TCT] is the result of a junior laboratory aiming at making the formalisation of Bachelor-level mathematics easier. It is born has an attempt to make a compromise between the classical axiomatisation of the standard library, which forbids any truly constructive result, and the constructive approach, which has a great impact on proofs, especially in Bachelor-level mathematics where the possibility to decide equality on real numbers and the law of Excluded Middle are pervasive.

This library is based on a constructive axiomatisation of real numbers [MP11] but does not rule out the use of classical rules of reasoning when needed.

Why we chose Coquelicot

Our goal is very close to the one of the junior laboratory at the origin of the COQTAIL library. Indeed, we want to make the formalisation in mathematics and especially in real analysis easier. Although constructive reasoning is a laudable concern and interesting subject, in particular with regards to real arithmetic [GNSW07], the common mathematical practice is to use classical reasoning when it is convenient. If we want to make formal mathematics closer to pen-and-paper proofs, we then have to allow for classical reasoning steps such as proof by contradiction (which is necessary to prove Lemma 2.2), which rules out the CoRN and MathClasses libraries.

Since Coquelicot extends Coq's standard library, solving on the way some of its issues, the choice is between COQTAIL and Coquelicot, i.e. between a constructive and a classical axiomatisation while allowing for additional axioms making classical reasoning possible. From

this perspective, COQTAIL seems to be the best choice since it does not forbid constructive results.

However, the COQTAIL library lacks the abstractions that make the success of the Coquelicot library. It is mainly focused on real functions while Coquelicot contains more general results, and convergence is expressed through $\varepsilon - \delta$ definitions, while Coquelicot benefits from the notion of *filter* we will discuss in Section 2.3.2. Since practicality and generality are to us more important concerns than logical foundations, which we can safely assert is also true of the average mathematician, we decided to use Coquelicot.

2.3.2 Filters for Real Analysis

Theorem 2.7 is a convergence theorem. As a consequence, we needed a good formalisation of convergence-related notions. The state of the art, in Coq [BLM15] but also in Isabelle/HOL [HIH13] and in Lean [LMCLD], indicates that *filters* form the appropriate notion to formalise convergence. Let us first give a few definitions about filters and convergence before describing our contributions.

Definitions

Filters are a generalisation of sequences introduced by Cartan [Car37b, Car37a] and later developed by the Bourbaki group [Bou71]. Their original definition corresponds to the notion of proper filter in Definition 2.7.

Definition 2.7 ((Proper) Filter)**.** A set of sets F is called a:

— filter if and only if it is non-empty, upward closed, and closed under intersection:

$$F \neq \emptyset, \quad \forall A, B.\ A \subseteq B \Rightarrow A \in F \Rightarrow B \in F \quad \text{and} \quad \forall A, B \in F.\ A \cap B \in F.$$

— proper filter if and only if it is a filter and it does not contain the empty set.



Remark

Since a filter F is non-empty and upward closed, it always contains the full set. This leads to the following definition of filters in Coquelicot, which is designed as a type class [SO08] in order to automatically infer the filter structure of given sets of sets.

```
Class Filter {T : Type} (F : (T -> Prop) -> Prop) := {
  filter_true : F (fun _ => True) ;
  filter_and : forall P Q : T -> Prop, F P -> F Q ->
    F (fun x => P x ∧ Q x) ;
  filter_imp : forall P Q : T -> Prop, (forall x, P x -> Q x) ->
    F P -> F Q
}.
```

Note that sets are represented in Coq as predicates, i.e. functions with values in `Prop`.

Example

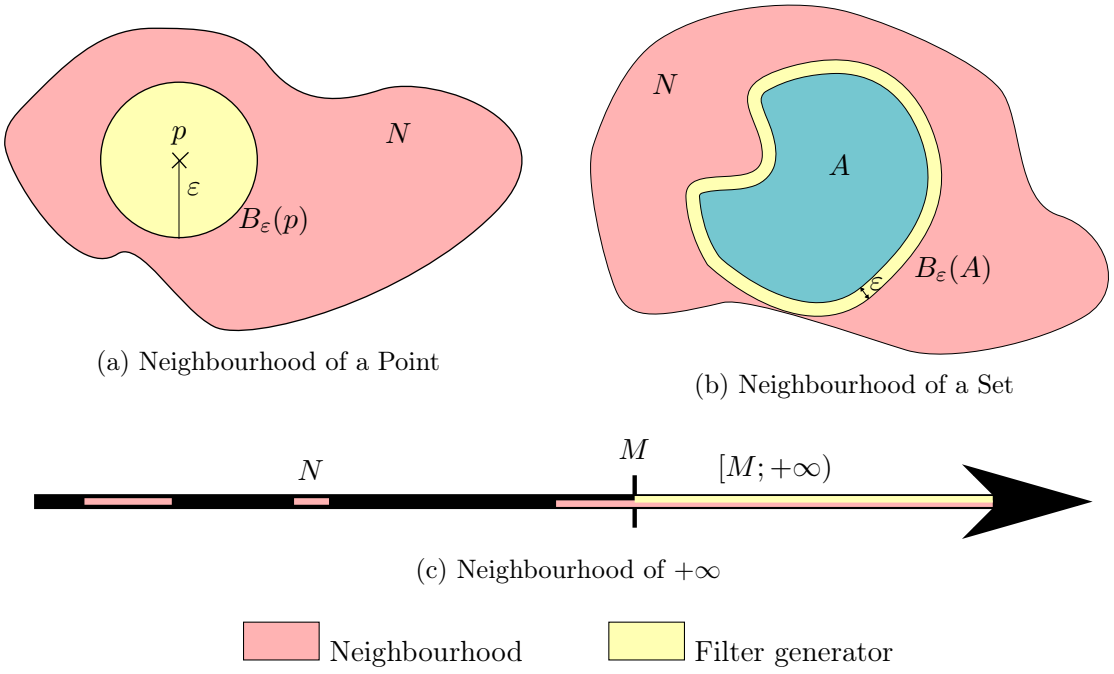
Filters are used in particular in analysis to represent neighbourhoods. Indeed, "being a neighbourhood" is a predicate that describes a filter.

For instance, the set $\{N \mid \exists \varepsilon > 0.\ B_\varepsilon(p) \subseteq N\}$ of neighbourhoods of a point p in a uniform space (see Figure 2.7a), where $B_\varepsilon(p)$ denotes the ball of centre p and radius ε, forms a filter denoted by `locally p` in Coquelicot.

Another kind of neighbourhoods that plays a role in the proof of Theorem 2.7 is the notion of neighbourhood of a set A (see Figure 2.7b). It generalises the notion of neighbourhood of a point in a straightforward way: instead of balls around the point, consider strips around the set. Formally, such a strip is the set of points p such that $d(p, A) < \varepsilon$ for a given $\varepsilon > 0$ (recall Definition 2.2 for the distance of a point to a set). We denote by $B_\varepsilon(A)$ such a strip (by analogy with balls) and by `locally_set A` the neighbourhood filter of the set `A`.

Finally, the set $\{N \mid \exists M.\ [M; +\infty) \subseteq N\}$ of "neighbourhoods of $+\infty$" (see Figure 2.7c) is a filter denoted by `Rbar_locally p_infty` in Coquelicot. In fact, the function `Rbar_locally` takes a point in $\overline{\mathbb{R}}$ (i.e. $\mathbb{R} \cup \{\pm\infty\}$) and returns a filter on $\mathbb{R}$: `locally p` if its argument `p` represents a real number, and the corresponding set of neighbourhoods of the infinity if it is either `p_infty` (for $+\infty$) or `m_infty` (for $-\infty$).

In all these examples, the neighbourhood filter is defined by taking the supersets of a family of generators (balls $B_\varepsilon(p)$ around points, strips $B_\varepsilon(A)$ around sets, or intervals $[M; +\infty)$ and $(-\infty; M]$ for the infinities). We will make formal such a way of defining filters in Section 5.3.2.



(a) Neighbourhood of a Point

(b) Neighbourhood of a Set

(c) Neighbourhood of $+\infty$

Figure 2.7 – Different Kinds of Neighbourhoods

An important construction for analysis is the image of a filter by a function. Given a function f and a filter F, the image of F by f, defined by $f(F) = \{B \mid f^{-1}(B) \in F\}$, is a filter, denoted by `filtermap f F` in Coquelicot.

Finally, filter inclusion is an important property. Indeed, this makes it possible to rephrase the $\varepsilon - \delta$ definition of limits into a more concise statement. The inclusion

$$f(\texttt{locally}(x)) \supseteq \texttt{locally}(y)$$

indeed unfolds to the elementary definition

$$\forall \varepsilon > 0.\ \exists \delta > 0.\ \forall z \in B_\delta(x).\ f(z) \in B_\varepsilon(y),$$

which represents the statement $f(z) \xrightarrow[z \to x]{} y$.

Set Notations

We developed notations in order to make the manipulation of filters closer to textbook mathematics. In particular, we define Mathematical Components-like notations [MCT] to denote set theoretic operations.

The type of sets over a type `T` is denoted by `set T`. Then, `set0`, `setT` and `[set p]` are respectively the empty set, the total set and the singleton $\{p\}$. Also, the notations `(A ‘&‘ B)`, `(A ‘|‘ B)`, `(~‘ A)` and `(A ‘\‘ B)` respectively denote the set intersection, union, complement and difference. We write `(A ‘<=‘ B)` for $A \subseteq B$ and `(A !=set0)` for $\exists p \in A$. The image and preimage of a set A by a function f are respectively denoted by `f @^‘ A` and `f @^-1‘ A`. Finally, we also introduce set comprehension notations `[set p | A]` (which is a typed alias for `(fun p => A)`) and the big operators `\bigcup_(i in A) F i` and `\bigcap_(i in A) F i` respectively denoting the union and intersection of families indexed by `A` (big operators were introduced in Coq by Bertot et al. [BGBP08]).

With these notations, the type class defining filters in Coquelicot can be rephrased in a more readable way, which is the one we chose to use in the library we will present in Part II [ACM+].

```
Class Filter {T : Type} (F : set (set T)) := {
  filterT : F setT ;
  filterI : forall P Q : set T, F P -> F Q -> F (P ‘&‘ Q) ;
  filterS : forall P Q : set T, P ‘<=‘ Q -> F P -> F Q
}.
```

It is also easy using these notations to define the strip of width ε around a set A as a generalisation of the notion of ball:

```
Definition ball_set {U : UniformSpace} (A : set U) (ε : posreal) :=
  \bigcup_(p in A) ball p ε,
```

where `posreal` is the type of positive real numbers.

Canonical Filters

Set-theoretic notations are not sufficient to make filter manipulation closer to pen-and-paper mathematics. Indeed, in Coquelicot we still have to represent the statement

$$f(x) \xrightarrow[x \to p]{} q, \text{ or equivalently } \lim_{x \to p} f(x) = q,$$

with the Coq term

```
filterlim f (locally p) (locally q),
```

where `filterlim` combines filter inclusion with the `filtermap` function as discussed earlier in this section.

Although Coquelicot introduces layers over filters to abbreviate convergence, they are still too restrictive to be practical. For example the predicate `is_lim f p q` is specialized to a real function `f` and to `p` and `q` in `Rbar` (which represents $\overline{\mathbb{R}}$) and is defined in terms of `filterlim`. Since we use other notions of convergence (convergence in a normed space, convergence to a set, recall the example page 28), adding more alternative definitions for approximately the same notion would only clutter the formalisation, so we decided to remove this extra layer. Instead, we provide a unique mechanism to infer which notion of convergence is required, by inspecting the form of the arguments and their types. We also define notations that trigger this mechanism and make the statements closer to hand-written mathematics.

First, what makes possible this factorisation is the fact that every type of convergence is expressed as filter inclusion involving the appropriate kind of neighbourhood filter. We denote by F `-->` G the reverse inclusion $F \supseteq G$. In fact, this notation triggers our inference mechanism so that it already provides a convenient notation for filter convergence (see Definition 2.8): the convergence of filter `F` to point `p` is denoted by `F --> p`.

Definition 2.8 (Filter Convergence)**.** We say that a filter F converges to a point p if and only if F contains the neighbourhood filter of p.

Remark

A filter may converge to several points. In fact, we may characterise Hausdorff spaces (see Definition 2.12) by the fact that filters do not converge to more than one point.

Then, in the case of function convergence, we also use the notation `f @ F` for the image of the filter `F` by the function `f`. Thanks to our inference mechanism, it is also possible to write `f @ p` to denote the image by `f` of the neighbourhood filter of `p`. Combining both notations, we can now write

```
f @ p --> q
```

instead of

```
filterlim f (locally p) (locally q).
```

This mechanism also adapts to sets (e.g. we write `f @ p --> A`) and to the infinities: with the notation `+oo` (respectively `-oo`) for `p_infty` (respectively `m_infty`), the appropriate instance of the `Rbar_locally` filter is inferred when we write `f @ +oo --> p` for "`f` converges to `p` when its argument goes to infinity", or `f @ p --> -oo` for "`f` diverges to $-\infty$ at point `p`".

Finally, although we do not use it in this part of our work, we also cast functions from `nat`, i.e. sequences, to the only sensible filter on $\mathbb{N}$ (the equivalent of `Rbar_locally p_infty` on `nat`,

named `eventually` in Coquelicot), so that one can write `u --> p` where `u : nat -> U` is a sequence.

Coq's coercion mechanism is not powerful enough to handle casts from an arbitrary term to an appropriate filter. Hence, the mechanism to automatically infer a filter from an arbitrary term and its type is implemented using canonical structures [Sai99, MT13].

We define a structure that recognises types whose elements could be cast to filters.

```
Structure canonical_filter {Y : Type} := CanonicalFilter {
  canonical_filter_type :> Type;
  _ : canonical_filter_type -> set (set Y)
}.
```

This structure associates to its first field, a type, a function that maps each of its elements to a filter on the parameter `Y` of the structure. For instance, on a uniform space `U`, the default filter is given by the `locally` function: each point is cast to its neighbourhood filter.

```
Canonical filter_uniform_space (U : UniformSpace) :=
  @CanonicalFilter U U (@locally U).
```

Moreover, having the parameter `Y` different from the `canonical_filter_type` field makes it possible to infer filters on `R` based on elements in `Rbar` through the `Rbar_locally` function.

```
Canonical filter_Rbar := @CanonicalFilter R Rbar Rbar_locally.
```

Our notations `F --> G` and `f @ p` respectively replace `F`, `G` and `p` with `[filter of F]`, `[filter of G]` and `[filter of p]`, where the `[filter of _]` notation hides the second projection of the `canonical_filter` structure. For example, for `p` of type `U` where `U` is a uniform space, `[filter of p]` is this second projection applied to `p`. For this to work, `U` must be convertible with `canonical_filter_type`, the first projection of the structure: unification triggers the inference of an instance of the canonical structure so that, thanks to the canonical instance `filter_uniform_space`, `[filter of p]` is actually `locally p`.

In the case where the user gives a filter instead of a point in a uniform space, our mechanism still makes it possible to use the same notation thanks to a second structure. This second structure recognises arrow types in order to associate filters to particular cases of functions: filters, but also sequences and sets, as we already discussed.

```
Structure canonical_filter_source {Z Y : Type} := CanonicalFilterSource {
  canonical_filter_source_type :> Type;
  _ : (canonical_filter_source_type -> Z) -> set (set Y)
}.
```

This structure associates the source type of an arrow type to a function mapping elements of this arrow type to filters on the parameter `Y` of the structure. Matching on the source type makes it possible to distinguish the different kinds of functions: for filters the source type is an arrow type with codomain `Prop` (i.e. the type of a set), for sequences it is `nat` and for sets it is a uniform space.

```
Canonical source_filter_filter (Y : Type) :=
  @CanonicalFilterSource Prop _ (_ -> Prop) (@id (set (set Y))).
```

```
Canonical eventually_filter (Z : Type) :=
  @CanonicalFilterSource Z _ nat (fun f => f @ eventually).

Canonical filter_set_uniform_space (U : UniformSpace) :=
  @CanonicalFilterSource Prop _ U locally_set.
```

We define a canonical instance of the `canonical_filter` structure that uses this second structure in order to complete the inference mechanism.

```
Canonical default_arrow_filter (Y Z : Type)
  (X : canonical_filter_source Z Y) :=
  @CanonicalFilter _ (X -> Z) (@canonical_source_filter _ _ X),
```

where `canonical_source_filter` is a name we gave to the second projection of the structure matching arrow types.

Remark

In fact, the `[filter of _]` notation uses the second projection of a third structure that recognises terms that could be cast to filters. This structure associates an element of a type with a filter on a second type.

```
Structure canonical_filter_on {X Y : Type} := CanonicalFilterOn {
  canonical_filter_term : X;
  _ : set (set Y)
}.
```

If the inference of an instance of this structure for a term `x` fails, a default instance triggers the inference of the `canonical_filter` structure on the type of `x`.

```
Canonical default_filter_term (Y : Type) (X : canonical_filter Y)
  (x : X) := @CanonicalFilterOn X Y x (canonical_type_filter x),
```

where `canonical_type_filter` is the second projection of the `canonical_filter` structure.

Actually, we ended up noticing that this structure always fails and the default instance is always used since we never declared instances for it: we thought particular terms would be associated to given filters but in all our use cases the type of the term was sufficient to determine the filter. As a consequence, the `canonical_filter_on` structure was removed in later versions of the mechanism and the `canonical_filter` structure became the default one (see Section 5.3.2).

2.3.3 Topological Notions

The other main ingredient of Theorem 2.7, besides convergence, is compactness. Indeed, the only solutions of the differential equation that are relevant in Theorem 2.7 have values in a compact set, which is an essential property in the proofs of Lemma 2.2 and Lemma 2.5.

Although many formalisations of topology express compactness using open covers (see Definition 2.9), we decided to experiment with a definition of compact sets using filters (see

Definition 2.11). This definition already appears in Cartan's work [Car37a] and happens to be particularly convenient in our settings. In fact, the filter-based definition of compactness involves the notion of clustering (see Definition 2.10), which is closely related to convergence and limit points. This notion can also be used to formalise closed sets.

Definition 2.9 (Compactness (Open Covers))**.** A set A is said to be compact if and only if for all family of open sets $(O_i)_{i\in I}$ that covers A (i.e. $A \subseteq \bigcup_{i\in I} O_i$), there exists a finite subset J of I such that the family $(O_i)_{i\in J}$ still covers A.

Definition 2.10 (Cluster)**.** A point p is a cluster point of the filter F if and only if each element of F intersects each neighbourhood of p.

We say that F clusters if it admits a cluster point.

Definition 2.11 (Compactness (Filters))**.** A set A is said to be compact if and only if every proper filter on A clusters in A.

We proved that Definition 2.9 and Definition 2.11 are equivalent. We use the filter-based definition of compactness by default in our formalisation, but we had sometimes to use the one based on open covers. We will explain why in our discussion about compact sets, but first we describe our formalisation of clustering.

Clustering

Our formalisation of clustering is a straightforward translation of Definition 2.10.

```
Definition cluster {U : UniformSpace} (F : set (set U)) (p : U) :=
  forall A B, F A -> locally p B -> A `&` B !=set0
```

To see the link with the limit points of a function `y`, consider the filter `y @ +oo`. The set of points to which `y @ +oo` clusters is exactly the positive limiting set of `y`, i.e. the set of limit points of `y` (recall Definition 2.5).

```
Definition pos_limit_set {U : UniformSpace} (y : R -> U) :=
  \bigcap_(eps : posreal) \bigcap_(T : posreal)
    [set p | Rlt T `&` (y @^-1` ball p eps) !=set0].

Lemma plim_set_cluster (U : UniformSpace) (y : R -> U) :
  pos_limit_set y = cluster (y @ +oo).
```

Note that we wrote an equality between sets, i.e. between functions with propositions as value. We used the axioms of functional extensionality (`funext`) and propositional extensionality (`propext`) to be able to prove this. This makes our code closer to textbook mathematics. This also makes it possible to use rewriting instead of applications of theorems to switch definitions, which combines well with the fact that proof scripts in the SSReflect tactic language [GMT15] rely heavily on the rewriting of equalities. All in all, our proofs are more natural and shorter than without these axioms.

```
Axiom funext : forall (T T' : Type) (f g : T -> T'), f =1 g -> f = g.

Axiom propext : forall (P Q : Prop), (P <-> Q) -> P = Q.
```

In the `funext` axiom, the notation `f =1 g` stands for `forall t, f t = g t`.

We also noticed that clustering can be used to define Hausdorff spaces (see Definition 2.12). Indeed, the contrapositive of Definition 2.12 admits a nice statement using clustering: if two points `p` and `q` are such that all their neighbourhoods intersect, i.e. the neighbourhood filter of `p` clusters to `q` (and vice-versa), then they are equal.

Definition 2.12 (Hausdorff Space)**.** A space U is said to be T_2 separable if and only if all distinct points in U can be separated by neighbourhoods, i.e. if and only if for all $p \neq q$ one can find two neighbourhoods A and B respectively of p and q such that A and B are disjoint (see Figure 2.8).

T_2 separable spaces are called Hausdorff spaces.

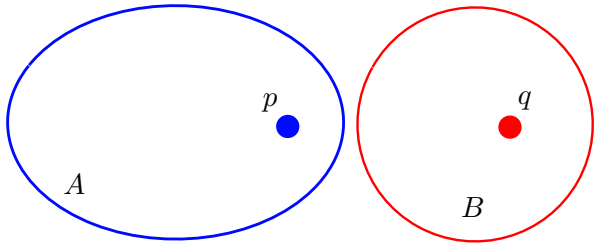


Figure 2.8 – Illustration of T_2 Separation

```
Definition hausdorff (U : UniformSpace) :=
  forall p q : U, cluster (locally p) q -> p = q.

Lemma hausdorffP (U : UniformSpace) :
  hausdorff U <-> forall p q : U, p <> q -> exists A B,
    locally p A ∧ locally q B ∧ forall r, ~ (A ‘&‘ B) r.
```

Compact Sets

Our translation of Definition 2.11 is straightforward, too: the filter `F` clusters in the set `A` if `A` and the set of cluster points of `F` have non-empty intersection.

```
Definition compact {U : UniformSpace} (A : set U) :=
  forall (F : set (set U)), F A -> ProperFilter F ->
  A ‘&‘ cluster F !=set0.
```

Remark

Note how the hypothesis "on `A`" from Definition 2.11 has been translated into "`A` is an element of `F`". This is possible thanks to the properties of filters: every filter on `A` is a filter base (we will make this precise in Definition 5.4) in `U` whose completion is a filter containing `A`, and every filter on `U` containing `A` defines a filter on `A` when restricted to sets contained in `A`.

Thanks to this, we do not have to consider compact spaces, which would require the use of topological notions such as the *subspace topology* (see Definition 3.2) and would

add complications. Indeed, the type classes `Filter` and `ProperFilter` of Coquelicot are defined on sets of sets on a type, i.e. on functions of type `(T -> Prop) -> Prop` for `T` of type `Type`, while in our context `A` is of type `U -> Prop`. Canonically transfering structures to subsets would then require wrapping functions into types, while our solution is simpler.

As a consequence, we never manipulate compact spaces but only compact sets: a compact space is a space in which the full set is compact.

We proved the equivalence between this definition and Definition 2.9 following the proof in Wilansky's textbook on topology [Wil08] (see our discussion on closed sets). We adapted Cano's formalisation [Can14] of Definition 2.9 to this end.

This notion of compact set is quite convenient to use to work with convergence and limit points: the only hard part is finding the right filter on the compact set and then the cluster point given by this hypothesis is usually the point one is looking for. However for other proofs this notion is quite complicated to use. Proving that a set is compact requires finding a cluster point for any abstract proper filter on this set, or going through a proof by contradiction. Moreover, to prove that any compact set is bounded, we had to go through the definition of compactness based on open covers. Indeed, the cluster point given by the filter-based definition does not give any information on the maximum norm of the elements of the set.

Closed Sets

In Coquelicot [BLM15], a set `A` is closed if and only if it contains all points the complement of `A` is not a neighbourhood of.

```
Definition closed {U : UniformSpace} (A : set U) :=
  forall p, ~ (locally p (~‘ A)) -> A p.
```

This basically translates to: `A` is closed if and only if the complement of `A` is open. Indeed, an open set is a neighbourhood of all its points. Thus, if `~‘ A` is open, the set of the points `~‘ A` is not a neighbourhood of is `A` and `A` is indeed the complement of an open set.

However, the notion of closure better fits our purposes. A point is in the closure of a set if all its neighbourhoods intersect the set. A set is closed if and only if its closure is included in it (the other inclusion always holds). This definition is (classically) equivalent to the one in Coquelicot.

```
Definition closure {U : UniformSpace} (A : set U) (p : U) :=
  forall B, locally p B -> A ‘&‘ B !=set0.

Definition is_closed {U : Uniformspace} (A : set U) := closure A ‘<=‘ A.

Lemma closedP (U : UniformSpace) (A : set U) : closed A <-> is_closed A.
```

This definition is more appropriate in our context than the one in Coquelicot because it gives us the only property of closed sets we needed (see the third paragraph in the proof of Lemma 2.5). Indeed, if the image of a function is included in a set A, then its set of limit points is included in the closure $\overline{A}$ of A. If moreover A is closed, then the set of limit points is included in A, since $\overline{A} = A$, which is the property we implicitely used in the proof of Lemma 2.5.

The notion of closure is also very practical because of its similarity with clustering: a filter clusters to a point if and only if this point is in the closure of each element of the filter.

```
Lemma clusterE (U : UniformSpace) (F : set (set U)) :
  cluster F = \bigcap_(A in F) (closure A).
```

This means that the definitions of clusters, Hausdorff spaces, compact sets and closed sets are very consistent. This makes it easier to prove for instance that compact sets in a Hausdorff space are closed or that Definition 2.9 and Definition 2.11 are equivalent. Indeed, we prove this equivalence by going through a third definition of compactness (see Definition 2.14), based on closed sets and the finite intersection property (see Definition 2.13).

Definition 2.13 (Finite Intersection Property)**.** A family of sets $(A_i)_{i\in I}$ has the finite intersection property if and only if for all finite subset J of I, the set $\bigcap_{i\in J} A_i$ is non-empty.

Definition 2.14 (Compactness (Closed Sets))**.** A set A is compact if and only if every family of closed sets of A with the finite intersection property has a non-empty intersection.

This definition is very close to the filter-based one, and we proved constructively that they are equivalent. Indeed, the set of all finite intersections in such a family is a filter base defining a proper filter which clusters. Conversely, the family of closures of the elements of a proper filter which clusters has the finite intersection property (the cluster point belongs to each closure). The equivalence between this third definition and Definition 2.9 is however classical. More precisely, both directions in this equivalence are proven by contraposition and classical steps are required to push negations under existential quantifiers and to remove double negations.

2.3.4 Formal Statement of the Invariance Principle

Now that we discussed the main ingredients of Theorem 2.7, we can focus on its formal statement. First, recall that, instead of functions on $\mathbb{R}^n$ we considered the ambient space to be any normed module over $\mathbb{R}$. It can in particular be an infinite-dimensional space. Let then `U` be such a space, which is considered to be an implicit argument in all this section.

Then, we need to express the property of being a solution of the differential equation given by Equation (2.1). We use the `is_derive` predicate from Coquelicot [BLM15] to express derivatives: `is_derive y t d` means that the derivative of `y` at time `t` is `d`. Since solutions are unique with respect to initial conditions, we use a single function `sol` of type `U -> R -> U` to represent them. For any initial condition `p`, `sol p` is a total function that represents the solution of Equation (2.1) starting at point `p`. A function `y` starting in `K` is a solution of Equation (2.1) if and only if it is equal to the function `sol (y 0)`, i.e. to the solution which has same initial condition.

```
Definition is_sol (y : R -> U) :=
  forall t, is_derive y t (F (y t)).

Hypothesis solP : forall y, K (y 0) -> is_sol y <-> y = sol (y 0).
```

This equality between functions, which we can use thanks to the axiom of functional extensionality, matches both the pen-and-paper and the SSReflect [GMT15] proof-styles.

Indeed, replacing a function by another one which is extensionally equal to it is frequent in mathematics and can be done here through rewriting, which takes an important part in proof scripts in the SSReflect tactic language. In addition, using the function `sol` in our statements instead of a function `y` together with the additional hypothesis `is_sol y` makes our statements more readable and simplifies our proofs.

The fact that the first argument of `sol` is the initial condition is expressed through the following hypothesis.

```
Hypothesis sol0 : forall p, sol p 0 = p.
```

Note that the combination of `sol0` and `solP` gives the existence and uniqueness of solutions. The continuity of solutions relative to initial conditions in the compact set K from Theorem 2.7 is expressed through a third hypothesis: when we fix the second argument of `sol`, we get a function which is continuous on K.

```
Hypothesis sol_cont : forall t, continuous_on K (sol^~ t).
```

However, we later noticed that our definition of the notion of solution is too restrictive. Indeed, some differential equations might have no meaning for some values of `t`. Solutions usually come with an interval of definition and we are interested in maximal solutions, i.e. the solutions whose interval of definition cannot be extended. In our case, since we are only interested in systems which have a physical interpretation, or in other terms in functions of time, we may restrict the domain of definition of solutions to the set of non-negative real numbers. The most straightforward way of doing so is to construct the type $\mathtt{R}^+$ of non-negative real numbers and to develop its theory in order to consider functions whose type is $\mathtt{R}^+$ `-> U`. We considered that the implementation cost of this representation would be too high, in particular because of the manipulation of dependent types this would require. Instead, we preferred keeping **total** functions on `R` with the constraint that the differential equation is only satisfied for non-negative times.

```
Definition is_sol (y : R -> U) :=
  forall t, 0 <= t -> is_derive y t (F (y t)).
```

Unfortunately, this definition is incompatible with `solP`. Indeed, this assumption is not satisfiable if we constrain the derivative of solutions only for non-negative times, since there would be (infinitely) many solutions with the same initial value (the values for negative times are basically free). In order to keep the benefits of the formulation of `solP`, we decided to change the notion of solution rather than to adapt `solP`. Once again, including the domain of definition of solutions into their type as suggested above would allow us to use an equality between functions as in `solP`, but this would require tedious manipulations of dependent types in order to develop a theory of these subtypes of `R`. We chose instead to fix the values of solutions for negative times in a way which does not constrain the differential equation. It is always possible since we only want to state properties on values for non-negative times. In order to keep the solutions derivable everywhere, we made them symmetric with regard to their initial value, i.e. $y(t) = 2y(0) - y(-t)$, which leads to the following definition in Coq.

```
Definition is_sol (y : R -> U) :=
  (forall t, t < 0 -> y t = 2 *: (y 0) - (y (- t))) ∧
  forall t, 0 <= t -> is_derive y t (F (y t)).
```

This definition does not make the function `sol` harder to use (there is only an assumption to discard). The only true drawback we encountered so far is that this definition significantly increases (around 50 additional lines) the size of the proof of Lemma 2.6. Indeed, we cannot directly use the uniqueness of solutions to prove that the function z is equal to the "shifted solution" $s \mapsto y(t+s)$: the "shifted solution", of the form $s \mapsto \mathrm{sol}_p(t+s)$ where $t \in \mathbb{R}^+$, may not be a solution any more since it may not be symmetric with regard to its initial value. It is first necessary to build a solution which coincides with the function $s \mapsto \mathrm{sol}_p(t+s)$ on $\mathbb{R}^+$ and to prove that it is indeed a solution. Only then we can use the uniqueness of solutions to conclude the proof.

With all these assumptions, we can finally give the formal statement of Theorem 2.7. As explained in Section 2.2.2, we proved the convergence of solutions of Equation (2.1) to the union of the positive limiting sets of the solutions starting in `K`, which can be expressed using clustering as discussed in Section 2.3.3.

```
Definition limS (A : set U) := \bigcup_(q in A) cluster (sol q @ +oo).
```

Recall that we require `K` to be compact and invariant. Both these hypotheses are used to prove the convergence of solutions to `limS K`.

```
Definition is_invariant (A : set U) :=
  forall p, A p -> forall t, 0 <= t -> A (sol p t).

Lemma cvg_to_limS (A : set U) : compact A -> is_invariant A ->
  forall p, A p -> sol p @ +oo --> limS A.
```

This is in fact an "easy" part of LaSalle's invariance principle. It is indeed sufficient for a function to ultimately [1] have values in a compact set in order for it to converge to the set of its limit points, hence to any superset of its positive limiting set.

```
Lemma cvg_to_pos_limit_set (y : R -> U) (A : set U) :
  (y @ +oo) A -> compact A -> y @ +oo --> cluster (y @ +oo).

Lemma cvg_to_superset (A B : set U) (y : R -> U) :
  A `<=` B -> y @ +oo --> A -> y @ +oo --> B.
```

The invariance of `K` is a strong way to force the solutions to ultimately have values in `K`. However, since in our proof of LaSalle's invariance principle we need to use the uniqueness of solutions for initial conditions which are values of solutions starting in `K`, the invariance of `K` is required anyway.

There are two other aspects to our version of LaSalle's invariance principle: `limS K` is invariant and it is a subset of the set of points `p` for which $\tilde{V}$`(p) = 0` (recall Definition 2.6).

The first point does not need any hypothesis: the positive limiting set of any solution starting in `K` is invariant (see Lemma 2.4), hence any union of such sets is invariant too.

```
Lemma invariant_pos_limit_set (p : U) :
  K p -> is_invariant (cluster (sol p @ +oo)).

Lemma invariant_limS (A : set U) : A `<=` K -> is_invariant (limS A).
```

1. This addition of "ultimately" generalises Lemma 2.2

The second point, corresponding to Lemma 2.5, requires the existence of a Lyapunov function, which is continuous on K, derivable along the trajectories of solutions and such that for all $p \in K$, $\tilde{V}(p) \leqslant 0$. We make use of the `Derive` function from Coquelicot to express $\tilde{V}$: `Derive f t` is the derivative of the real function `f` at time `t`.

```
Lemma stable_limS (V : U -> R) :
  continuous_on K V ->
  (forall p t, K p -> 0 <= t -> ex_derive (V \o (sol p)) t) ->
  (forall (p : U), K p -> Derive (V \o (sol p)) 0 <= 0) ->
  limS K `<=` [set p | Derive (V \o (sol p)) 0 = 0].
```

2.4 Related Work

We divide our references into three categories corresponding to the different topics we discussed in this chapter: related work on stability analysis (not necessarily in a proof assistant) in Section 2.4.1, related work on the formalisation of topology in Section 2.4.2 and related work on the formalisation of differential equations in Section 2.4.3.

2.4.1 Related Work on Stability Analysis

Let us first discuss formalisations on stability and Lyapunov functions and then other generalisations of LaSalle's invariance principle [LaS60, LaS76].

Chan et al. [CRLM16] used a Lyapunov function to prove in Coq the stability of a particular system. They have however no proof of a general stability theorem.

Mitra and Chandy [MC08] formalised in PVS stability theorems using Lyapunov-like functions in the particular case of automata. This is quite different from our work since they work in a discrete settings.

Herencia-Zapana et al. [HJO+12] took another approach to stability proofs: stability proofs using Lyapunov functions, under the form of Hoare triples annotations on C code implementing controllers, are used to generate proof obligations for PVS.

We are definitely not the first to generalise LaSalle's invariance principle. We decided to prove a version of the principle which is close to the original statement but several generalisations were designed to make it available in more complex settings.

Chellaboina et al. [CLH99] weakened further the regularity hypothesis on the Lyapunov function at the cost of sign conditions and a boundedness hypothesis on the Lyapunov function along the trajectories.

Barkana [Bar14, Bar17] restricted the hypotheses on the Lyapunov function to hypotheses along bounded trajectories in order to generalise LaSalle's invariance principle to non-autonomous systems.

Mancilla-Aguilar and García [MG06] generalised LaSalle's invariance principle to switched autonomous systems by adding further conditions related to switching, but removed the conditions of existence and uniqueness of solutions and of continuity of the solutions relative to initial conditions by working on a set of admissible trajectories.

Fischer et al. [FKD13] also weakened the hypotheses on the solutions of a non-autonomous system by using a generalised notion of solution.

2.4.2 Related Work on the Formalisation of Topology

Several formalisations in topology already exist: in Coq [Can14], in PVS [Les07], in Isabelle/HOL [HIH13], in Lean [LMCLD] or in Mizar [Dar90, PD90] for instance. All of them, except the one in Lean, express compactness using open covers.

We adapted Cano's formalisation [Can14] of compactness based on open covers for our proof of equivalence with the filter-based definition. We could not use it directly since it relies on the `eqType` structure of the Mathematical Components library and Coquelicot's structures [BLM15] are not based on this structure.

Note that in the work of Hölzl et al. [HIH13] there is a definition of compactness in terms of filters which is slightly different from ours: a set A is compact if for each proper filter on A there is a point $p \in A$ such that a neighbourhood of p is contained in the filter. This is a bit less convenient to use than clustering since one cannot choose the neighbourhood. In Lean, compactness is also defined using filters, in a way which is equivalent to ours. To our knowledge, our work is the first attempt to exploit the filter-based definition of compactness to get simple proofs on convergence.

We must also mention Coquelicot's definition of compactness, which is based on gauge functions, and Coq's topology library by Schepler [Scha]. Both are unfortunately unusable in our context: Coquelicot's definition is specialised to $\mathbb{R}^n$ while we are working on an abstract normed module, and Schepler's library does not interface with Coquelicot, since it redefines filters for instance. Schepler's library contains a proof of equivalence between the filter-based and open covers-based definitions of compactness, which is very close to ours. However, these definitions concern topological spaces whereas, as mentioned in Section 2.3.3, we focus on subsets of such spaces without referring to the subspace topology.

2.4.3 Related Work on the Formalisation of Differential Equations

Differential equations have been studied by Boldo et al. [BCF+13] and by Immler and Hölzl [IH12] with a special focus on numerical approximation schemes. In particular, Immler later extended his work to formally verify an algorithm computing numerical approximations of solutions of an ordinary differential equation [Imm18]. We are however interested in the qualitative analysis of the equations: we prove properties on the solutions without finding them analytically and without computing approximations.

The work of Immler and Hölzl is not entirely focused on numerical approximations: they prove the qualitative Cauchy-Lipschitz Theorem, which constitutes the basis for later work by Immler and Traut [IT16, IT19], who formalise the theory of the flow of an ordinary differential equation. Our function `sol` is akin to the flow but, in absence of a proper theory of the flow in Coq, we do not bother with existence intervals of solutions and assume solutions are always defined for any time.

Other formalisations of the Cauchy-Lipschitz Theorem include Maggesi's one in HOL Light [Mag18], which proves only the local version of this theorem, and the one by Makarov and Spitters [MS13], which is based on the CoRN library [CGW04] while we work in an inherently classical settings (recall our discussion from Section 2.3.1).

We thought about proving the Cauchy-Lipschitz Theorem using the fixed-point theorem formalised by Boldo et al. [BCF+17] in order to have an appropriate theory of the flow and to generalise Theorem 2.7 (considering limits at the upper bound of the existence interval instead

of limits at infinity). However, the proof of this theorem relies on the theory of integration so that we decided instead to wait for our new library for analysis in Coq (see Part II) to be more complete in order to have better tools for this proof. We are currently collaborating with Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi and Pierre-Yves Strub to develop the theory of integration in our library.

Part I
Case Study: the Inverted Pendulum

CHAPTER 3

SWING-UP OF THE INVERTED PENDULUM

In this chapter we present our formalisation in the Coq proof assistant of the proof of soundness of a control function for the inverted pendulum designed by Lozano et al. [LFB00]. This function, presented as a force applied to the dynamical system of the pendulum as discussed in Section 1.2.2, performs the swing-up of the pendulum.

We first finish in Section 3.1 the description of the dynamical system we started in Section 1.2.2. We focus in particular on the control function. Then, we give the mathematical proof of soundness of this control function in Section 3.2, which in fact boils down to the stability of the controlled system. Finally, we discuss notable aspects of the formalisation in Section 3.3 and we mention related work in Section 3.4.



Warning

While working on this formalisation, we found a few errors in the mathematical proof by Lozano et al.. Their statement is however true, since we managed to correct the proof. We will highlight in ***slanted boldface*** the places where the errors were made and describe our way to correct them.

This chapter is based on our publication on the topic [Rou18]. All code snippets come from our on-line repository [CRa].

3.1 The Dynamical System

We first recall in Section 3.1.1 the dynamical system we described in Section 1.2.2 and we clearly explain what is the soundness property which was proven by Lozano et al. [LFB00]. As a second step we give in Section 3.1.2 a reformulation of the differential equation, which is

the one we used in our formalisation, and we present the ideas of Lozano et al. for the design of the control function.

3.1.1 The Dynamical System and its Control Challenge

For the reader's convenience, we repeat Figure 1.5 as Figure 3.1 and Equation (1.6) as Equation (3.1). We recall that q denotes the state (or configuration) of the system, which is given by the position x of the cart on the horizontal line and the angle θ the pole forms with the vertical line. We also denote by l the length of the weighted pole, m and M the respective masses of the weight and of the cart, g the gravitational acceleration on Earth and fctrl the control force.

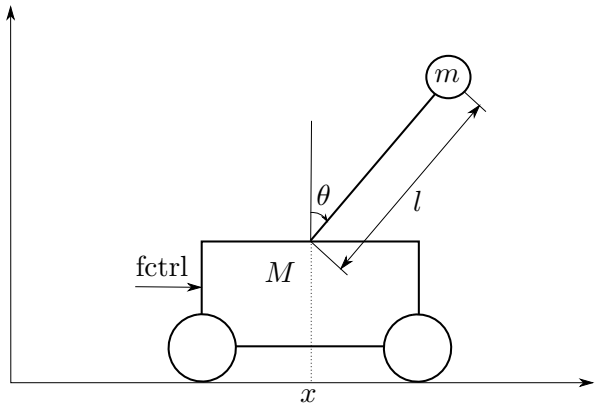


Figure 3.1 – The Inverted Pendulum with Annotations (repeated)

$$M(q)\ddot{q} + C\left(q,\dot{q}\right)\dot{q} + G\left(q\right) = \tau\left(q,\dot{q}\right), \tag{3.1}$$

where

$$q = \begin{pmatrix} x \\ \theta \end{pmatrix}, \qquad C\left(q,\dot{q}\right) = \begin{pmatrix} 0 & -ml\dot{\theta}\sin\theta \\ 0 & 0 \end{pmatrix},$$

$$M(q) = \begin{pmatrix} M+m & ml\cos\theta \\ ml\cos\theta & ml^2 \end{pmatrix}, \qquad G(q) = \begin{pmatrix} 0 \\ -mgl\sin\theta \end{pmatrix},$$

$$\tau\left(q,\dot{q}\right) = \begin{pmatrix} \text{fctrl}\left(q,\dot{q}\right) \\ 0 \end{pmatrix}.$$

Remark

This time we explicitly give the parameters on which τ depends. Note that the position x of the cart will only appear in the expression of the control function. Indeed, the physics of the (free) system does not depend on the position of the cart.

As we explained in Section 1.1.3, the goal of the control function here is to bring this system to the unstable equilibrium of the free pendulum. More precisely, we want to prove that the (equilibrium of the) controlled inverted pendulum is asymptotically stable (recall Definition 1.1). Instead of proving the convergence of the pendulum to its upper equilibrium, Lozano et al. [LFB00] prove the convergence to a trajectory which converges to the equilibrium, called *homoclinic orbit*. This trajectory is characterised by the following differential equation:

$$\frac{1}{2}ml^2\dot{\theta}^2 = mgl\,(1-\cos\theta)\,. \tag{3.2}$$

Moreover, they want the cart to stop at its starting point, so that on this trajectory we also want to have $x = 0$ and $\dot{x} = 0$. The soundness of the control function is thus expressed as the asymptotic stability of the set of points which lie on the homoclinic orbit (recall Definition 2.3). This stability property can be stated as in Theorem 3.1.

Theorem 3.1 (Soundness of the Control Function)**.** *For some set K of starting positions and for a well-chosen control function* fctrl*, all solutions of Equation* (3.1) *starting in K converge to the homoclinic orbit described by Equation* (3.2)*, together with the property that $x = 0$ and $\dot{x} = 0$, when time goes to infinity.*

As detailed in Section 2.3.4, we represent the solutions of a differential equation using a function `sol` which takes the initial position of the inverted pendulum as input and computes the trajectory of the system. Here, it also depends on the control function but for simplicity we do not display this dependency in our notations. We can then state Theorem 3.1 in Coq as follows.

```
Lemma cvg_to_homoclinic_orbit (p : 'rV[R]_5) :
  K p -> sol p @ +oo --> homoclinic_orbit.
```

Here, `'rV[R]_5` is the type for vectors in $\mathbb{R}^5$ from the Mathematical Components library [MCT]. We explain in Section 3.1.2 why the state space is $\mathbb{R}^5$ and not $\mathbb{R}^4$ (two dimensions for q and two for $\dot{q}$). The definitions of the control function and of the set K of valid starting positions and the Coq description of the homoclinic orbit also come in Section 3.1.2.

3.1.2 The System We Actually Formalised

A major tool in the proof of stability by Lozano et al. [LFB00] is LaSalle's invariance principle [LaS60, LaS76]. However the invariance principle requires the dynamical system to be described by a first-order autonomous differential equation (recall Section 1.2.1). In order to rewrite Equation (3.1) as such an equation, it is necessary to consider a state that contains more information:

$$z = \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} x \\ \dot{x} \\ \cos\theta \\ \sin\theta \\ \dot{\theta} \end{pmatrix}.$$

Remark

Lozano et al. chose to split the information given by θ into the information of its sine and cosine. This simplifies the differential equation since only these values appear in it. This has the side effect of making the dimension of the state space to be five instead of four.

It is then possible to transform Equation (3.1) into the following equation:

$$\dot{z} = \text{Fpendulum}\,(z)\,, \tag{3.3}$$

where, for all $p = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$,

$$\text{Fpendulum}\,(p) = \begin{pmatrix} p_1 \\ \frac{mp_3\left(lp_4^2 - gp_2\right) + \text{fctrl}(p)}{M + mp_3^2} \\ -p_3 p_4 \\ p_2 p_4 \\ \frac{(M+m)gp_3 - p_2\left(mlp_4^2 p_3 + \text{fctrl}(p)\right)}{l\left(M + mp_3^2\right)} \end{pmatrix}.$$

With these notations, the homoclinic orbit is simply the set of points p such that

$$p_0 = 0 \text{ and } p_1 = 0 \text{ and } \frac{1}{2}ml^2p_4^2 = mgl\,(1 - p_2)\,.$$

In Coq [CDT19], this set is defined in a very similar way thanks to the use of notations for set comprehension and for the access to components of vectors.

```
Definition homoclinic_orbit :=
  [set p : 'rV[R]_5 | p[0] = 0 ∧ p[1] = 0 ∧
   (1 / 2) * m * (l ^ 2) * (p[4] ^ 2) = m * g * l * (1 - p[2])].
```

However, with Equation (3.3), we lose some pieces of information. For instance, since we work with points in $\mathbb{R}^5$, we forget that z_2 is a cosine. It will then be necessary to pose constraints on the initial value $z(0)$ in order to keep the lost information as an invariant. For example, the equation $z_2^2(t) + z_3^2(t) = 1$ is to be proven for any time t. This is the role of the set K.

The definition of K is a consequence of the use of LaSalle's invariance principle: it is the compact set that is used in the invariance principle. The challenge behind the use of LaSalle's invariance principle is to find an appropriate Lyapunov function V so that being an invariant subset of $\left\{p \in K \mid \tilde{V}(p) = 0\right\}$ grants the desired properties. In our case, the desired property is to be a subset of the set of points described by `homoclinic_orbit`.

In order to achieve this goal, Lozano et al. choose an energy approach. The energy of the system is

$$E\,(q, \dot{q}) = \frac{1}{2}\dot{q}^T M\,(q)\,\dot{q} + mgl\,(\cos\theta - 1)\,, \tag{3.4}$$

so that at the unstable equilibrium the energy is null. Conversely, when $E(q, \dot{q}) = 0$ and $\dot{x} = 0$, Equation (3.4) becomes Equation (3.2), thus the equation of the homoclinic orbit. Hence, it is sufficient to find a Lyapunov function V such that LaSalle's invariance principle proves the convergence to a set where $E = 0$, $\dot{x} = 0$ and $x = 0$.

The choice of V affects the choice of the compact set K, in the sense that the sign condition on $\tilde{V}$ imposes constraints on K. As mentioned in Section 2.2.1, the Lyapunov function V is minimised along the trajectories hence there is the obvious choice

$$V(q, \dot{q}) = \frac{k_E}{2} E^2(q, \dot{q}) + \frac{k_v}{2} \dot{x}^2 + \frac{k_x}{2} x^2 \tag{3.5}$$

with k_E, k_v and k_x positive constants.

With the notations of Equation (3.3), Equation (3.5) becomes

$$V(z) = \frac{k_E}{2} E^2(z) + \frac{k_v}{2} z_1^2 + \frac{k_x}{2} z_0^2.$$

This choice of Lyapunov function explains the definition of the control function by Lozano et al.. Indeed, an important assumption to be proven is that $\tilde{V}(p) \leqslant 0$ for all point p in the compact set K still to be defined. Computation shows that when z is a solution of Equation (3.3), the derivative of $V \circ z$ is

$$z_1 \left(\text{fctrl}(z) \left(k_E E(z) + \frac{k_v}{M + m z_3^2} \right) + \frac{k_v m z_3 \left(l z_4^2 - g z_2 \right)}{M + m z_3^2} + k_x z_0 \right),$$

so that the control function

$$\text{fctrl}(p) = \frac{k_v m p_3 \left(g p_2 - l p_4^2 \right) - \left(M + m p_3^2 \right) \left(k_x p_0 + k_d p_1 \right)}{k_v + \left(M + m p_3^2 \right) k_E E(p)}$$

for k_d a positive constant gives

$$\tilde{V}(p) = -k_d p_1^2 \leqslant 0.$$

This choice of control function imposes several constraints on the system and on the compact set K. First of all, this function needs to be well-defined and smooth in K in order to have the existence and uniqueness of solutions of Equation (3.3) and the continuity of solutions relative to initial conditions on K. Moreover, it should not drive the system outside of its domain of definition. Thus, we need to prove that for any solution z starting in K we have

$$k_v + \left(M + m z_3^2 \right) k_E E(z) \neq 0.$$

Knowing that z_3 represents a sine (hence $z_3^2 \leqslant 1$) and that K should be invariant, it will be sufficient to have for all $p \in K$

$$|E(p)| < \frac{k_v}{k_E (M + m)}.$$

Then, to avoid converging to the stable equilibrium of the pendulum, where the energy is $E(p) = -2mgl$, we want to ensure that $|E(p)| < 2mgl$ for $p \in K$. Overall, we want that

$$|E(p)| < b = \min\left(\frac{k_v}{k_E(M+m)}, 2mgl\right).$$

It is then sufficient to have $V(p) < B = \frac{k_E b^2}{2}$ for $p \in K$ in order to prove the constraint on E. ***This constant corrects the one given by Lozano et al., where*** k_E ***was forgotten***. Finally, as remarked before, since we forget that p_2 represents a cosine and p_3 a sine, we also need to ensure that $p_2^2 + p_3^2 = 1$, which leads to the following definition of the compact set K:

$$K = \left\{p \in \mathbb{R}^5 \mid p_2^2 + p_3^2 = 1 \text{ and } V(p) \leqslant k_0\right\}$$

where k_0 is a constant such that $k_0 < B$.

3.2 Stability Proof

In order to prove the stability of this inverted pendulum, we have first to show that the system satisfies the hypotheses of Theorem 2.7 (see Section 3.2.1). Then, we prove in Section 3.2.2 that this theorem indeed grants the convergence of the inverted pendulum to the homoclinic orbit given by Equation (3.2) together with the property that the cart's position and speed are null. We follow here the proof by Lozano et al. [LFB00], briefly giving the missing justifications for the use of LaSalle's invariance principle, and highlighting the places where the proof by Lozano et al. was erroneous. We sum up the errors in the proof in Section 3.2.3.

3.2.1 Verification of the Hypotheses of LaSalle's Invariance Principle

The first step in the verification of the inverted pendulum is to check that the system is well-defined. Looking at the form of Fpendulum in Equation (3.3), it is sufficient to prove that for all $p \in K$, $M + mp_3^2 \neq 0$ and $\text{fctrl}(p)$ is well-defined. The first point is in fact true for any p. The control function is well-defined at points such that its denominator is non zero. Following the reasoning in Section 3.1.2, we get that fctrl is well-defined on K. Hence, the system is well-defined on K.

Then, we need to prove the hypotheses of Theorem 2.7. First, **we admit the existence and uniqueness of solutions of Equation** (3.3) **and the continuity of solutions relative to initial conditions on** K. More precisely, we admit the existence of a function sol that satisfies the hypotheses described in Section 2.3.4 for the system defined by Fpendulum. A way to prove these properties is to use the Cauchy-Lipschitz Theorem. However, the only formalisation of this theorem in the Coq proof assistant [CDT19] we are aware of is the one by Makarov and Spitters [MS13]. It is based on the CoRN library of constructive real numbers [CGW04], while our formalisation of the inverted pendulum is based on the Coquelicot library [BLM15], which extends Coq's standard library on classically axiomatised real numbers [May01].

Then, the set K is compact, since it is closed and bounded and we are in finite dimension (we work in $\mathbb{R}^5$). The remaining hypotheses of Theorem 2.7 are

Hypothesis 1 K is invariant.

Hypothesis 2 For all $p \in K$, $V \circ \text{sol}_p$ is derivable at any time.

Hypothesis 3 For all $p \in K$, $\tilde{V}(p) \leqslant 0$.

While proving the invariance of K, ***we found a circular dependency between two properties in the proof by Lozano et al., none of them being proven in the end***. Indeed, given a point $p \in K$, the goal is to prove that for all non-negative t, $\mathrm{sol}_p(t) \in K$. This decomposes into two parts:

$$(\mathrm{sol}_p(t))_2^2 + (\mathrm{sol}_p(t))_3^2 = 1$$

and

$$(V \circ \mathrm{sol}_p)(t) \leqslant k_0.$$

The former is easy but in order to prove the latter, the authors argue that $V \circ \mathrm{sol}_p$ is non-increasing, which concludes the proof since $\mathrm{sol}_p(0) = p$ and $p \in K$. However, to show that $V \circ \mathrm{sol}_p$ is non-increasing, it is necessary that $\mathrm{fctrl} \circ \mathrm{sol}_p$ is well-defined, hence that sol_p stays in K.

We managed to eliminate this circular dependency and to prove the three remaining hypotheses by proving Lemma 3.2.

Lemma 3.2. *For all $p \in K$, for all time t, the derivative at time t of $V \circ \mathrm{sol}_p$ is $-k_d(\mathrm{sol}_p(t))_1^2$.*

Let us first check that Lemma 3.2 indeed implies the three remaining hypotheses. First, **Hypothesis 2** is an obvious corollary of Lemma 3.2. Then, **Hypothesis 3** is true because by definition $\tilde{V}(p)$ is the derivative at time 0 of $V \circ \mathrm{sol}_p$. Finally, Lemma 3.2 implies that if $p \in K$, then $V \circ \mathrm{sol}_p$ is non-increasing, which proves **Hypothesis 1** as discussed above.

In the remainder of this section, we explain how we managed to eliminate the circular dependency in order to prove Lemma 3.2. In fact, given a time t and a point $p \in K$, the derivative of $V \circ \mathrm{sol}_p$ at time t has indeed the desired value as soon as fctrl is well-defined at point $\mathrm{sol}_p(t)$. Instead of proving that fctrl is well-defined on K, we noticed that the reasoning in Section 3.1.2 proves Lemma 3.3, which states that fctrl is well-defined on a set which is larger than K.

Lemma 3.3. *The control function* fctrl *is well-defined at the points p such that $p_3^2 \leqslant 1$ and $V(p) < B$.*

Then, since $\left\{p \in \mathbb{R}^5 \mid p_2^2 + p_3^2 = 1\right\}$ is invariant, it is possible to show that Lemma 3.3 implies Lemma 3.4.

Lemma 3.4. *For all $p \in K$ and all time t, the derivative at time t of $V \circ \mathrm{sol}_p$ is $-k_d(\mathrm{sol}_p(t))_1^2$* **if** *$(V \circ \mathrm{sol}_p)(t) < B$.*

Using Lemma 3.4, it is sufficient to prove Lemma 3.5 in order to get Lemma 3.2.

Lemma 3.5. *For all $p \in K$ and all time t, $(V \circ \mathrm{sol}_p)(t) < B$.*

It might seem that we are doing the same reasoning as Lozano et al.: we use the expression of the derivative of $V \circ \mathrm{sol}_p$ in order to prove that $V \circ \mathrm{sol}_p$ stays below a given bound. Moreover, proving that the bound is correct is necessary to prove the validity of the expression for this derivative. However, there is here a crucial difference with the previous goal: now we have a **strict** inequality in Lemma 3.5 compared to the **large** inequality in the definition of K.

This allows us to do the following proof. Consider s the greatest lower bound of the set A of times at which the condition is not satisfied: $A = \{t \in \mathbb{R}^+ \mid B \leqslant (V \circ \mathrm{sol}_p)(t)\}$ and $s = \inf A$. The goal is to prove that $s = +\infty$. Since $0 \leqslant s$, it is sufficient to prove that s cannot be finite, which we do by contradiction.

In the case where s is finite, we show that s is a minimum, which implies $B \leqslant (V \circ \mathrm{sol}_p)(s)$. This is where having a **large** inequality in the definition of s is important. Indeed, if s is not a minimum, then we have the **strict** inequality $(V \circ \mathrm{sol}_p)(s) < B$. Thus, since $V \circ \mathrm{sol}_p$ in continuous at point s, there exists ε such that for all $t \in (s-\varepsilon; s+\varepsilon)$, we have $(V \circ sol_p)(t) < B$. In particular this means that $s + \frac{\varepsilon}{2}$ is a lower bound of A which is greater than s, which contradicts the definition of s.

We can also prove that s is positive. Moreover, for all time $0 < t < s$, since we know that $(V \circ \mathrm{sol}_p)(t) < B$ by definition of the greatest lower bound, we also know thanks to Lemma 3.4 that the derivative of $V \circ \mathrm{sol}_p$ at time t is $-k_d (\mathrm{sol}_p(t))_1^2$. Since $V \circ \mathrm{sol}_p$ is also continuous on $[0; s]$, Rolle's Theorem proves that for some $t \in (0; s)$

$$\frac{(V \circ \mathrm{sol}_p)(s) - (V \circ \mathrm{sol}_p)(0)}{s - 0} = -k_d (\mathrm{sol}_p(t))_1^2 \leqslant 0.$$

Thus, $(V \circ \mathrm{sol}_p)(s) \leqslant V(p)$, which is a contradiction since $V(p) < B \leqslant (V \circ \mathrm{sol}_p)(s)$.

3.2.2 Convergence to the Homoclinic Orbit

Theorem 2.7 applied to the inverted pendulum proves the convergence of any solution of Equation (3.3) starting in K to the set

$$L = \bigcup_{\substack{y \text{ solution of Equation (3.3)} \\ \text{starting in } K}} \Gamma^+(y).$$

The goal is now to prove that L is included in the homoclinic orbit characterised by Equation (3.2) with the additional property that for all $p \in L$, $p_0 = 0$ and $p_1 = 0$. As mentioned in Section 3.1.2, it is sufficient to prove that for all $p \in L$,

Goal 1 $p_0 = 0$.

Goal 2 $p_1 = 0$.

Goal 3 $E(p) = 0$.

Let then p be in L. Since L is an invariant subset of the set $\left\{p \in K \mid \tilde{V}(p) = 0\right\}$ by Theorem 2.7, we know that the derivative of $V \circ \mathrm{sol}_p$ at time 0 is null. Using Equation (3.2) we prove that the derivative function of $V \circ \mathrm{sol}_p$ is the identically zero function. As proven in Section 3.2.1, this derivative function is also $t \mapsto -k_d (\mathrm{sol}_p(t))_1^2$. From this, we deduce Lemma 3.6.

Lemma 3.6. *For all $p \in L$,*

1. *$t \mapsto (\mathrm{sol}_p(t))_1$ is the identically zero function.*
2. *$t \mapsto (\mathrm{sol}_p(t))_0$ is constant.*
3. *$E \circ \mathrm{sol}_p$ is constant.*
4. *$t \mapsto ((\mathrm{Fpendulum} \circ \mathrm{sol}_p)(t))_1$ is the identically zero function.*

In particular, from point 1 of Lemma 3.6 at time 0 we get **Goal 2**. From all this, we can also derive the important equations

$$\forall t, k_E \left(E \circ \mathrm{sol}_p\right)(t) \left(\mathrm{fctrl} \circ \mathrm{sol}_p\right)(t) + k_x \left(\mathrm{sol}_p(t)\right)_0 = 0, \tag{3.6}$$

and

$$\forall t, \left(\mathrm{sol}_p(t)\right)_3 \left(g \left(\mathrm{sol}_p(t)\right)_2 - l \left(\mathrm{sol}_p(t)\right)_4^2\right) = \frac{\left(\mathrm{fctrl} \circ \mathrm{sol}_p\right)(t)}{m}. \tag{3.7}$$

We know by Lemma 3.6 that $E \circ \mathrm{sol}_p$ is constant (equal to $E(p)$). Either this constant is zero or it is not. If it is zero, i.e. **Goal 3** is true, from Equation (3.6) at time 0 we prove **Goal 1**, which ends the proof.

If $E(p) \neq 0$, we want to derive a contradiction by proving that it implies that $E(p) = 0$, or equivalently (thanks to Lemma 3.6) that $E \circ \mathrm{sol}_p$ is the identically zero function. From Equation (3.6) and points 2 and 3 of Lemma 3.6 we prove Lemma 3.7.

Lemma 3.7. *For all $p \in L$ such that $E(p) \neq 0$, $\mathrm{fctrl} \circ \mathrm{sol}_p$ is constant.*

Up to this point, we followed the proof by Lozano et al. [LFB00]. ***But then the authors present an erroneous (and unnecessary) proof that*** $\mathrm{fctrl} \circ \mathrm{sol}_p$ ***is the identically zero function. They draw from that, again through an erroneous proof, the conclusion that*** $E \circ \mathrm{sol}_p$ ***is the identically zero function.***

In order to understand their error, it is important to know that Lemma 3.8 is a significant tool for the remaining parts of the proof. If an equation e between two functions is true on an interval, we will call *derivative of e* the equation obtained from e using Lemma 3.8.

Lemma 3.8. *Let I be an interval which is* **not reduced to a point** *and f and g be two real functions differentiable on I.*

If for all $t \in I$, $f(t) = g(t)$, then for all $t \in I$, $\dot{f}(t) = \dot{g}(t)$.

Taking the derivative of Equation (3.7) and using Lemma 3.7, we get the following equation

$$\forall t, \left(\mathrm{sol}_p(t)\right)_4 \left(3g \left(\left(\mathrm{sol}_p(t)\right)_2^2 - \left(\mathrm{sol}_p(t)\right)_3^2\right) + C \left(\mathrm{sol}_p(t)\right)_2\right) = 0, \tag{3.8}$$

where C is a constant depending on $E(p)$.

Then, for a given time t, if $\left(\mathrm{sol}_p(t)\right)_4 \neq 0$, then we get from Equation (3.8) that

$$3g \left(\left(\mathrm{sol}_p(t)\right)_2^2 - \left(\mathrm{sol}_p(t)\right)_3^2\right) + C \left(\mathrm{sol}_p(t)\right)_2 = 0. \tag{3.9}$$

The authors then proceed to take the derivative of this new equation, which is incorrect because an equation has to be valid on an interval for one to be allowed to take its derivative. In this particular case, it is still possible to take the derivative of Equation (3.9) because it is true on a small interval containing t. Indeed, when $\left(\mathrm{sol}_p(t)\right)_4 \neq 0$, since sol_p is continuous one can find a positive real number ε such that for all $s \in (t - \varepsilon; t + \varepsilon)$, we have $\left(\mathrm{sol}_p(s)\right)_4 \neq 0$. However, they repeat several times this error by considering that when a component of sol_p is null at some time t, the derivative of this component also must be null at time t. This cannot be corrected by such a simple argument of continuity.

While looking for a way to correct the proof by Lozano et al., we found a way to simplify the proof that $E \circ \mathrm{sol}_p$ is the identically zero function. It does not require proving that $\mathrm{fctrl} \circ \mathrm{sol}_p$

is the identically zero function, only that it is constant (in order to use Equation (3.8)), which we already know by Lemma 3.7.

The first step is to prove the following equation, which we obtain from Equation (3.7) and Equation (3.11):

$$\forall t, \left(\text{Fpendulum}\left(\text{sol}_p\left(t\right)\right)\right)_4 = \frac{g}{l}\left(\text{sol}_p\left(t\right)\right)_3. \tag{3.10}$$

$$\forall t, \left(\text{sol}_p\left(t\right)\right)_2^2 + \left(\text{sol}_p\left(t\right)\right)_3^2 = 1. \tag{3.11}$$

Equation (3.10) has several consequences. In particular, we can prove Lemma 3.9, from which we deduce Lemma 3.10 using Equation (3.1), Equation (3.11) and Equation (3.10).

Lemma 3.9. *For all $p \in L$ such that $E\left(p\right) \neq 0$, the component functions $t \mapsto \left(\text{sol}_p\left(t\right)\right)_2$ and $t \mapsto \left(\text{sol}_p\left(t\right)\right)_3$ are constant.*

Lemma 3.10. *For all $p \in L$ such that $E\left(p\right) \neq 0$,*

1. *$t \mapsto \left(\text{sol}_p\left(t\right)\right)_4$ is the identically zero function.*
2. *$t \mapsto \left(\text{sol}_p\left(t\right)\right)_3$ is the identically zero function.*
3. *For all time t, $\left(\text{sol}_p\left(t\right)\right)_2$ is either 1 or -1.*

From Equation (3.4), point 1 of Lemma 3.6 and point 1 of Lemma 3.10, we know that

$$\forall t, \left(E \circ \text{sol}_p\right)\left(t\right) = mgl\left(\left(\text{sol}_p\left(t\right)\right)_2 - 1\right). \tag{3.12}$$

Hence, in order for $E \circ \text{sol}_p$ to be the identically zero function, it is sufficient to prove that

$$\forall t, \left(\text{sol}_p\left(t\right)\right)_2 = 1.$$

By point 3 of Lemma 3.10, it is sufficient to prove that $\left(\text{sol}_p\left(t\right)\right)_2 = -1$ is not possible. When $\left(\text{sol}_p\left(t\right)\right)_2 = -1$, with Equation (3.12) we get the equation

$$\left(E \circ \text{sol}_p\right)\left(t\right) = -2mgl,$$

which contradicts the condition $\left|E\left(p\right)\right| < b$ from Section 3.1.2 ($p \in K$ and $E \circ \text{sol}_p$ is constant).

All in all, this gives a (correct) simpler proof that all solutions of Equation (3.1) starting in K converge to a set L included in the homoclinic orbit characterised by Equation (3.2), with the additional property that for all $p \in L$, $p_0 = 0$ and $p_1 = 0$. We end this section with an idea of the proof of Lemma 3.9.

Both points of Lemma 3.9 can be proven using Lemma 3.11. Indeed, using Equation (3.10) and Equation (3.11) in Equation (3.8) and its derivative, and discussing whether $\left(\text{sol}_p\left(t\right)\right)_4$ is null or not, we get two polynomials of degree two whose coefficients do not depend on t and such that $\left(\text{sol}_p\left(t\right)\right)_2$ is a root of one of them. Thus, the component function $t \mapsto \left(\text{sol}_p\left(t\right)\right)_2$ can have only a finite number of different values and, by Lemma 3.11, is constant. Similarly, using Equation (3.11) we prove that the component function $t \mapsto \left(\text{sol}_p\left(t\right)\right)_3$ can have only two different values, since $t \mapsto \left(\text{sol}_p\left(t\right)\right)_2$ is constant. It is thus constant by Lemma 3.11.

Lemma 3.11. *Let I be an interval and f be a real function.*

If f is continuous on I and if f can only take a finite number of different values on I, then f is constant on I.

3.2.3 Summary of the Corrected Errors

The stability proof by Lozano et al. [LFB00] contains three kinds of errors.

1. There is a forgotten constant in the definition of the compact set K.
The set K is defined as

$$K = \left\{p \in \mathbb{R}^5 \mid p_2^2 + p_3^2 = 1 \text{ and } V(p) \leqslant k_0\right\},$$

where k_0 is a constant such that

$$k_0 < k_E \frac{\left(\min\left(\frac{k_v}{k_E(M+m)}, 2mgl\right)\right)^2}{2}.$$

The multiplication by k_E in the above inequality was forgotten by Lozano et al..

2. There is a circular dependency between two properties: they are proven equivalent but none of them is actually proven.

In order to prove that K is invariant, Lozano et al. use the fact that for any $t \geqslant 0$, the derivative of $V \circ \mathrm{sol}_p$ at time t is $-k_d \left(\mathrm{sol}_p(t)\right)_1^2$. However, in order to prove this property, they use the fact that fctrl is well-defined at point $\mathrm{sol}_p(t)$ because this point belongs to K, hence they use the invariance of K.

We corrected this error by proving that fctrl is well-defined on a set which is larger than K and that solutions that start in K stay in this larger set.

3. Some computation steps rely on the derivative of equations that are only true at a given point.

This error happens several times in the proof by Lozano et al.. Sometimes the computation step is still valid because an argument of continuity provides an interval on which the equation is true, thus allowing us to take its derivative, but this is not always the case.

We corrected this error by writing a new proof that does not follow the same reasoning.

3.3 Formalisation of the Stability Proof

In this section, we give details on the formalisation of the proof we discussed in Section 3.2. We start by giving some context on the libraries and data structures we use (see Section 3.3.1). Then we explain in Section 3.3.2 why we chose to formalise topological spaces, which were missing from Coquelicot [BLM15]. Finally, we discuss in Section 3.3.3 how we designed a way to automatically compute derivatives and differentials in order to simplify proofs.

3.3.1 On the Choice of Data Structures

Our formalisation builds on the one we presented in Chapter 2, hence the main library we are using is Coquelicot. Coquelicot contains a hierarchy of algebraic and topological structures (recall Figure 2.6) and equips Coq's type `R` of real numbers with all these structures. In our case, since the system described in Section 3.1.2 is in $\mathbb{R}^5$, we have first to choose a data type which represents vectors in $\mathbb{R}^5$ and to equip this type with Coquelicot's structures.

We follow the example of Paşca's work on multivariate analysis [Paş08, Paş11] and use the structure of vectors from the Mathematical Components library [MCT] (see [Paş08] for a discussion on the different possibilities for representing $\mathbb{R}^n$ in Coq). A point in $\mathbb{R}^n$ is thus represented as an element of the type `'rV[R]_n` of row vectors on `R` of length `n`.

Mathematical Components contains a different formalisation of vectors than the inductive type `vector` we presented in the introduction of this thesis. Indeed, a row vector of length `n` is a matrix with one line and `n` columns. Matrices with `m` rows and `n` columns are represented in Mathematical Components as functions from `'I_m * 'I_n` to the type of coefficients, where `'I_p` is the type of natural numbers `k` such that `k < p`. Internally, functions whose domain is finite (as it is the case here) are represented as tuples of length the cardinal of the domain, i.e. sequences bundled with a proof that their length is this cardinal. The intuition is the following: each element e of the domain gets a number n and the image of e is the element of rank n in the sequence.

The main difference with Paşca's work is that we have access to the Coquelicot library (and that the Mathematical Components library has evolved since then), so that we work in a more convenient framework to do multivariate analysis where we can reuse many theorems already proven.

Still, we have to equip the type `'rV[R]_n` with Coquelicot's structures so that they are automatically inferred where needed. We can prove that when a type `T` has a certain structure, the type `'rV[T]_n` canonically inherits this structure. This is the matter of a few 500 lines, using straightforward definitions such as componentwise addition for the abelian group structure, or componentwise multiplication by an element of a ring A for the A-module structure.

However, difficulties arise with the definition of a norm over the type of vectors. A natural choice is the infinity norm

$$\left\|(v_i)_{i\in I}\right\|_\infty = \max_{i\in I} \|v_i\| .$$

Since the maximum operator on real numbers `Rmax` is a binary operator, we have to iterate it in order to compute the infinity norm of a vector in $\mathbb{R}^n$. We can easily define it using Mathematical Components' library for iterated operators over an indexed set (big operators [BGBP08]): the notation `\big[op/e]_i f i` defines the iteration of the operator `op` over the family `f` and the (assumed neutral) element `e`.

```
Definition vnorm (x : 'rV[T]_n) := \big[Rmax/0]_i (norm (x ord0 i)).
```

In this definition, `(x ord0 i)` is the `i`$^{\text{th}}$ component of the vector `x`: recall that row vectors are matrices with one row and `n` columns. Thus, the first argument of the function which represents `x` is a point in `'I_1`, i.e. the set of natural numbers n such that $n < 1$: the only possibility is 0, represented by `ord0`. In our case, the neutral element is `0` since norms are non-negative numbers.

The main difficulty with this definition comes from the fact that some theorems on big operators require algebraic structure on the operator, in particular the existence of a neutral element. This algebraic structure is inferred thanks to canonical structures [Sai99, MT13]. However, the maximum operator on real numbers does not canonically admit such a structure: with no further assumption the maximum operator does not have a neutral element.

Here, since we only consider non-negative numbers, `0` is a neutral element. Paşca [Paş11] suggests two possibilities to get round this issue: to build the type of non-negative real numbers, equip it with the right structure, and use the theorems in Mathematical Components' library for big operators, or to define a new maximum operator which has the right structure by definition and such that it coincides with `Rmax` on non-negative numbers and move from one operator to the other through rewriting.

The first possibility mentioned by Paşca seems to be the best way in the long-term but developing a theory of non-negative real numbers in Coq is not so obvious. For instance, if $\mathtt{R}^+$ is a new type built on top of the type `R`, it inherits from `R` its monoid structure but proving this in Coq requires a redefinition of addition as an operation on $\mathtt{R}^+$ and a proof that this new addition is associative and that the `0` of $\mathtt{R}^+$ (which is not the same as the `0` of `R`) is an identity for this operation. This entails a duplication of the code we would like to avoid.

We experimented with the second possibility, which was Paşca's choice, but we eventually opted for a third choice of implementation which revealed to be shorter: to prove again the theorems from the Mathematical Components library that require some algebraic structure, instantiated on real numbers together with the additional hypothesis that we only consider families of non-negative numbers. This is also a duplication of the code, but way shorter than in the first possibility. We also believe these theorems are not limited to non-negative real numbers and can actually be generalised: our proofs illustrate that the essential property of `e` in `\big[op/e]_i f i` in the duplicated theorems is idempotence (i.e. `op e e = e`), which is weaker than neutrality.

3.3.2 Topological Spaces

Topology is an important topic of this work. Indeed, we heavily rely on filters, we use LaSalle's invariance principle which is a theorem of convergence and we work with compact sets. However, we do not really commit to doing topology: we never manipulate *topological spaces* (see Definition 3.1). A reason for this is that the Coquelicot library does not deal with topological spaces. In fact, the topological structure which is at the base of Coquelicot's hierarchy is the structure of uniform space and not all topological spaces are uniform spaces [Wil08].

Definition 3.1 (Topological Space)**.** A set T is said to be a topological space if and only if it is equipped with a family $(O_i)_{i\in I}$ of sets, called open sets, such that

— T and $\emptyset$ are open sets.

— for every subfamily $(O_i)_{i\in J}$, the union $\bigcup_{i\in J} O_i$ is open.

— for every finite subfamily $(O_i)_{i\in J}$, the intersection $\bigcap_{i\in J} O_i$ is open.

The family $(O_i)_{i\in I}$ is called a topology on T.

In particular, as explained in Section 2.3.3, we speak of **compact sets** instead of **compact spaces**, thus avoiding the use of the *subspace topology* (see Definition 3.2).

Definition 3.2 (Subspace Topology)**.** Let T be a topological space with its family of open sets $(O_i)_{i\in I}$ and $S \subseteq T$.

The subspace topology on S is defined by the family $(O_i \cap S)_{i\in I}$.

In our formalisation of the inverted pendulum, as explained in Section 3.2.1, we prove that the set K is compact by proving that it is closed and bounded: this is called the Heine-Borel Theorem. A closed and bounded set is compact in a finite-dimensional space because it is a closed subset of a compact set (the finite product of segments defined by its bound). We have thus to prove that a finite product of compact sets in $\mathbb{R}$ is compact.

We decided to formalise a more general property, called Tychonoff's Theorem, which admits a simple proof thanks to filters, although it requires the axiom of choice in the form of Zorn's Lemma (we used the version of Zorn's Lemma contained in Schepler's small library on set theory [Schb]).

Tychonoff's Theorem states that any product of compact topological spaces is compact. We can rephrase this theorem as: any product of compact sets is compact. Its proof uses the definition of compactness in terms of ultrafilters (see Definition 3.3 and Lemma 3.12).

Definition 3.3 (Ultrafilter)**.** A set of sets F is an ultrafilter if and only if it is a proper filter and it is maximal for set inclusion, i.e. for every proper filter G, if $F \subseteq G$ then $F = G$.

Lemma 3.12. *A set A is compact if and only if every ultra filter on A converges in A (recall Definition 2.8).*

What makes the proof simple is the fact that, when given a product space $T = \prod_{i \in I} T_i$, an ultrafilter F on T and $p \in T$, F converges to p if and only if for all $i \in I$, the ultrafilter $F_i = \{\pi_i(A) \mid A \in F\}$ converges to $\pi_i(p)$, where π_i is the canonical projection to T_i. When each T_i is compact, we have by definition the convergence of F_i to some p_i and thus we can build a p to which F converges.

However, this proof is not possible if we stay in the world of uniform spaces because of the underlying topology. In a product of topological spaces, the natural topology is the product topology (see Definition 3.6), which is defined from the weak topology (see Definition 3.4) and the supremum topology (see Definition 3.5).

Definition 3.4 (Weak Topology)**.** Let S be a set, T be a topological space with its family of open sets $(O_i)_{i \in I}$ and f be a function from S to T.

The family $\left(f^{-1}(O_i)\right)_{i \in I}$ defines a topology on S, called the weak topology by f.

Definition 3.5 (Supremum Topology)**.** Let $(\Phi_i)_{i \in I}$ be a family of topologies on the same set T.

The supremum topology of $(\Phi_i)_{i \in I}$ is the smallest topology (for set inclusion) on T that contains every open set of each topology Φ_i for $i \in I$.

Definition 3.6 (Product Topology)**.** Let $(T_i)_{i \in I}$ be a family of topological spaces.

The product topology on the product space $\prod_{i \in I} T_i$ is the supremum topology of the family of weak topologies by each projection π_i on T_i.

The proof we gave actually exploits the properties of the product topology. However, in uniform spaces, we are stuck with the uniform topology (see Definition 3.7).

Definition 3.7 (Uniform Topology)**.** Let U be a uniform space.

Balls in U induce a topology on U as follows: a set A is open if and only if for all $p \in A$, there exists $\varepsilon > 0$ such that $B_\varepsilon(p) \subseteq A$.

The uniform topology is semi-metrisable (i.e. we can find a semi-metric that induces it through balls for this semi-metric), while the supremum topology of an uncountable family of semi-metrisable topologies might not be semi-metrisable [Wil08]. Thus, the uniform topology on an uncountable product of uniform spaces does not correspond to the product topology.

Since we wanted to prove Tychonoff's theorem in its full generality, even though we did not really need it, we decided to formalise topological spaces and prove Tychonoff's theorem using the product topology. Since the uniform topology on the finite product $\mathtt{U}^n$, where `U` is a uniform space, is the same topology as the product topology of the uniform topology on `U`, we were able to use this formalisation to prove the Heine-Borel Theorem.

There are two minor differences between Definition 3.1 and our formalisation thereof: finite intersections are reduced to intersections of arity 2 (n-ary intersections can be obtained by iteration) and we do not require the empty set to be open, since it is provable (take an union indexed on the empty set).

```
Record mixin_of (T : Type) := Mixin {
  open : set T -> Prop ;
  op_setU : forall (I : eqType) (f : I -> set T),
    (forall i, open (f i)) -> open (\bigcup_i f i) ;
  op_setI : forall (A B : set T), open A -> open B -> open (A `&` B) ;
  op_setT : open setT
}.
```

The family of open sets is represented as a predicate on sets. We use for the domains of indices the `eqType` structure that comes from the Mathematical Components library and represents types that have a decidable equality: it is indeed important to be able to compare two indices. The name `mixin_of` comes from the way we implement hierarchies (we will give more details in Section 5.1.2).

3.3.3 Automatic Computation of Differentials

While formalising the mathematics involved in the proof of soundness of the control function, we noticed that several times we had a simplified form for the differential of a function. To prove that it is indeed a differential we performed the same steps.

1. First prove that this differential can be written in a more expanded way.
2. Then use the rules of differentiation to prove that this expanded form is the differential of a given function.

For example, the derivative function of $E \circ \mathrm{sol}_p$ is

$$t \mapsto (\mathrm{sol}_p\,(t))_1\,(\mathrm{fctrl} \circ \mathrm{sol}_p)\,(t)\,.$$

But in order to prove this fact it is first necessary to put this function under the form

$$t \mapsto \mathrm{d}\,E_{\mathrm{sol}_p(t)}\,(\mathrm{Fpendulum}\,(\mathrm{sol}_p\,(t)))\,,$$

where $\mathrm{d}\,E_q$ is the expanded form of the differential of E at point q. Then, applying the rule for the differentiation of a composition of functions (also known as chain rule), we have to prove that $\mathrm{d}\,E_{\mathrm{sol}_p(t)}$ is the differential of E at point $\mathrm{sol}_p\,(t)$ and that $\mathrm{Fpendulum}\,(\mathrm{sol}_p\,(t))$ is

the derivative of sol_p at time t. The first goal is similarly proven, successively using different rules for differentiation (e.g. the first rule will be the one for the addition of two functions, see Equation (3.4) for the form of E).

For the first step (expanding functions to put them in the right form), the proof of equality is already quite automated thanks to reflection-based decision procedures for the equality of terms such as `ring` [GM05] or `field` [DM01] (see Section 9.1 for an introduction about reflection). For the second step (using the rules of differentiation), the only automation provided in Coquelicot is for the case of functions from $\mathbb{R}$ to $\mathbb{R}$ [LM12].

We later learnt that reversing the order between these two steps is possible thanks to the `evar_last` tactic from Coquelicot and allows for a slight improvement. When the goal is of the form $P\ p_1 \dots p_n$, with P a predicate and $p_1 \dots p_n$ its parameters, this tactic replaces the last parameter with an existential variable $?v$, and the goal with the two goals $P\ p_1 \dots ?v$ and $?v = p_n$. This allows us to use the rules of differentiation without having to provide the expanded form for the differential, since it will be inferred from the constraints these rules impose on $?v$. Our contribution is a way to automate this inference.

The automation of the computation of a differential for a given function is done by means of type classes [SO08]. We keep a data base of differentiation rules thanks to a type class `diff` encapsulating the `filterdiff` predicate from Coquelicot: `filterdiff f F df` means that `df` is the differential of `f` at the neighbourhood defined by `F`. In the remainder of this section, `K` is a ring equipped with an absolute value (`AbsRing` structure in Coquelicot) and `U` and `V` are two normed modules over `K`.

```
Context {K : AbsRing} {U V : NormedModule K}.

Class diff (f : U -> V) (F : set (set U)) (df : U -> V) :=
  diff_prf : filterdiff f F df.
```

When we want to prove that a function `df` is the differential of a given function `f`, we apply the following lemma.

```
Lemma diff_eq (f f' df : U -> V) (F : set (set U)):
  diff f F f' -> f' = df -> diff f F df.
```

Thanks to the SSReflect tactic language [GMT15], `f'` is introduced as an existential variable and type class inference is triggered in a seamless way. Type class inference will then automatically compute for us the function `f'` and prove the assumption `diff f F f'`. Then we can prove that `f'` is equal to `df` using for instance the axiom of functional extensionality and the `ring` and `field` tactics.

In order for type class inference to succeed, it is necessary to have a well-stocked data base of differentiation rules. We turned rules from Coquelicot into instances of the class `diff`. For example, we give the rules for the differential of constant functions, for the identity function, and for the sum of two functions (the proofs are direct applications of the corresponding lemmas from Coquelicot).

```
Instance diff_const (p : V) (F : set (set U)) :
  Filter F -> diff (fun _ => p) F (fun _ => 0).

Instance diff_id : diff id F id.
```

```
Instance diff_plus (f g df dg : U -> V) (F : set (set U)) :
  Filter F -> diff f F df -> diff g F dg ->
  diff (fun p => (f p) + (g p)) F (fun p => (df p) + (dg p)).
```

Sometimes, the form of the differential is not interesting because it is sufficient to prove that this differential exists. It is possible to prove automatically the existence of differentials by triggering type class inference thanks to the following lemma.

```
Lemma ex_diff (f df : U -> V) (F : set (set U)) :
  diff f F df -> ex_filterdiff f F.
```

We also have a similar mechanism for derivatives of functions whose domain is a ring with an absolute value. We define a type class `deriv` encapsulating the `is_derive` predicate from Coquelicot. We prove two lemmas to trigger type class inference whenever we want to prove that an expression is the derivative of a given function at some point or to prove that such a derivative exists.

```
Class deriv (f : K -> V) (x : K) (df : V) :=
  deriv_prf : is_derive f x df.

Lemma deriv_eq (f : K -> V) (x : K) (df' df : V) :
  deriv f x df' -> df' = df -> deriv f x df.

Lemma ex_deriv (f : K -> V) (x : K) (df : V) :
  deriv f x df -> ex_derive f x.
```

And we also build a data base of rules easily extracted from the Coquelicot library.

```
Instance deriv_const (p : V) (x : K) : deriv (fun _ => p) x 0.

Instance deriv_id (x : K) : deriv id x 1.

Instance deriv_plus (f g : K -> V) (x : K) (df dg : V) :
  deriv f x df -> deriv g x dg ->
  deriv (fun y => (f y) + (g y)) x (df + dg).
```



Example

Let us illustrate our methodology on a very trivial example: we want to prove that the derivative at point `x` of the function `fun y => p + y`, for `p` a point in `K`, is `1`. The goal is the following one:

```
is_derive (fun y => p + y) x 1.
```

In Coquelicot, one would prove this goal as follows: first apply the `evar_last` tactic to get the two goals

```
is_derive (fun y => p + y) x ?d,
?d = 1,
```

where `?d` is an existential variable representing the derivative at point `x` of our function.

On the first goal, using Coquelicot's equivalent of `deriv_plus` we have to prove the two goals

```
is_derive (fun _ => p) x ?d1,
is_derive id x ?d2,
```

where `?d1` and `?d2` are two fresh existential variables and `?d` has been partially instantiated into `?d1 + ?d2`.

Then, the equivalent of `deriv_const` (respectively `deriv_id`) closes the first (respectively second) goal. The full instantiation of `?d` is thus `0 + 1` and the last open goal is

```
0 + 1 = 1,
```

which is easily proven using the rules of the ring structure. For more complex expressions, the `ring` tactic [GM05] may be used.

With our inference mechanism, the first part of this proof is automated. On the goal

```
is_derive (fun y => p + y) x 1,
```

Lemma `deriv_eq` yields the two same goals as the `evar_last` tactic (the first one being encapsulated in the `deriv` class), but also triggers type class inference and automatically solves the first goal, yielding the goal

```
0 + 1 = 1,
```

which may be solved as before.

3.4 Related Work

We divide our references into two categories: related work on dynamical systems and control theory (see Section 3.4.1) and related work on the formalisation of mathematics (multivariate analysis and topology) in Section 3.4.2.

3.4.1 Related Work on Dynamical Systems and Control Theory

Several formalisations on dynamical systems and control theory already exist. Important tools in this domain are the KeYmaera prover [PQ08] and its successor, KeYmaera X [FMQ+15]. They operate however on a quite different domain since they are based on differential dynamic logic. Moreover, KeYmaera and KeYmaera X use Mathematica as a trusted oracle for quantifier elimination.

Anand et al. [AK15] developed a framework to build certified programs in Coq for robots, ROSCoq. They followed an approach which is similar to ours: first define the physics of the system using differential equations and then prove properties on it. An important difference with our work is that they use the constructive real numbers from the CoRN library [CGW04]. They are thus able to run the programs they certified, whereas it is impossible to carry out computation using the real numbers from Coq's standard library. It is however possible to use such real numbers as a tool in the formal verification of executable functions [BCF+13, BRT18].

The VeriDrone project [MRAL16] is another framework in Coq which is designed for cyber-physical systems. However, it is based on a different logic (it uses an embedding of

linear temporal logic in Coq) and also trusts external tools (SMT solvers). In this framework, Chan et al. [CRLM16] use a Lyapunov function to prove the stability of a particular system. They have however no proof of a general stability theorem and thus have to do a direct proof.

Herencia-Zapana et al. [HJO+12] take another approach to stability proofs: stability proofs using Lyapunov functions, under the form of Hoare triples annotations on C code, are used to generate proof obligations for PVS. By this means, they can directly prove properties on implementations instead of proving them on models of the systems.

3.4.2 Related Work on the Formalisation of Mathematics

Our contributions to the formalisation of mathematics in this chapter cover two domains: multivariate analysis and topology.

Multivariate Analysis

About multivariate analysis, besides Paşca's work [Paş08, Paş11] which we mentioned before, we have access to another formalisation in Coq of $\mathbb{R}^n$. In the Coquelicot library [BLM15], a type for $\mathbb{R}^n$ is defined thanks to a recursive function iterating cartesian product. It is used to define a type for matrices, but Coquelicot only contains few theorems on these stuctures and the Mathematical Components-like approach is more convenient in practice. Harrison, in his work on $\mathbb{R}^n$ in HOL Light [Har05, Har13], also comes to the conclusion that functions from a finite type of cardinality n to $\mathbb{R}$ are a good way to represent points in $\mathbb{R}^n$ but the limitations of HOL Light force him to use "encoding tricks". Isabelle/HOL also inherits this formalisation from HOL Light.

Concerning differentials, we decided to use type classes to add automation in spite of the existence in Coquelicot of a reflexive (i.e. based on reflection, which we introduce in Section 9.1) tactic computing derivatives [LM12, BLM12]. The main issue with this tactic is that it works only on functions from $\mathbb{R}$ to itself, while we have at some point to compute the differential of a function whose domain is $\mathbb{R}^n$ (e.g. we need to compute the differential of E in order to compute the derivative of $E \circ \mathrm{sol}_p$). We can also mention the formal proof of the automatic differentiation algorithm O∂yssée by Mayero [May02], which is also limited to functions from $\mathbb{R}$ to itself.

Topology

Topological spaces were formalised in most of our references from Section 2.4.2. Let us briefly comment on the ones in Coq.

Cano's work [Can14] contains a structure of topological space which is slightly more complex than ours because Cano does not use extensionality axioms. Another (anecdotal) difference is that he uses sets of sets to represent families in his definition of topological spaces whereas we use indexed families.

The library for topology by Schepler [Scha] is based on his library for set theory [Schb], which also represent families as sets of sets. This library also contains a proof of Tychonoff's Theorem. However, it has its own definitions of filters and uniform spaces and does not interface well with Coquelicot. For instance, Schepler's filters are Coquelicot's proper filters and Schepler's formalisation of uniform spaces implies that these are metric spaces while Coquelicot's uniform spaces are not necessarily metric. Moreover, in Schepler's library the

definition of compactness concerns topological spaces whereas, as mentioned before, we focus on subsets of such spaces without referring to the subspace topology.

The only other formalisations of Tychonoff's Theorem we know about are the one in Mizar by Skorulski [Sko01], using open covers, and the one in Lean [LMCLD], using ultrafilters.

Part I
Case Study: the Inverted Pendulum

CHAPTER 4

ASSESSMENT OF THE FORMALISATION

The goal of this case study was to determine limitations in existing libraries for analysis in Coq [CDT19] and to sketch solutions to overcome these limitations. We focused on the Coquelicot library [BLM15] for reasons explained in Section 2.3.1.

In this work, we designed new tools that make the formalisation of mathematics using Coquelicot smoother (see Section 4.1) and that open the door to new projects that will undoubtedly suggest other improvements through the challenges they raise (see Section 4.2). However, our work also revealed other issues that the tools we presented do not solve (see Section 4.3).

4.1 Improvements on the Existing

In order to achieve the formalisation of the inverted pendulum, we had to extend Coquelicot with a few concepts (e.g. compact sets or topological spaces) or to reformulate parts of Coquelicot in a classical settings (closed sets, mainly). These extensions put aside, our contribution to the formalisation of mathematics using Coquelicot is two-fold: on one hand, we designed tools that improve the user's experience with Coquelicot (see Section 4.1.1) and, on the other hand, we started experimenting with the use of Coquelicot in other fields of mathematics (see Section 4.1.2).

4.1.1 A Smoother Experience with Coquelicot

As we explained in the introduction of this thesis, we are concerned with the ease of use of the available formalisation tools and libraries. We believe our contributions described in Section 2.3.2 and Section 3.3.3 are an important step towards this goal.

Indeed, our notations that exploit our filter inference mechanism make statements easier to read (and faster to write), since they look closer to pen-and-paper notations. In particular, these notations hide filters that are usually left implicit: in mathematics, we talk about the limit of a function at a given point p, which implicitly means that we use the neighbourhood filter of p. If one wants for instance to talk about the limit when p is excluded (e.g. the limit of $x \mapsto \frac{1}{|x|}$ at 0), a specific notation (not formalised yet) is used in pen-and-paper mathematics. This specific notation indicates that another filter is used.

Moreover, on top of this better phrasing of mathematical statements in Coquelicot, our mechanism for the automatic computation of differentials and derivatives also makes proofs closer to pen-and-paper ones. Indeed, when one wants to prove that the differential of a function has a given value, the first step is usually to start from an "obvious" value and then to apply simplifications in order to get the expected result. A value is "obvious" because one can compute it without writing anything: computation steps are usually implicit. Our data base makes it possible to automatically compute such an "obvious" value, and existing reflection-based tactics also help proving the expected result through automatic computation steps.

4.1.2 Using Coquelicot in other Fields of Mathematics

In front of the diversity of libraries of the fields of mathematics and of the libraries that formalise parts of these fields, the best strategy is to try to reuse and combine as much as possible, when it is possible, what already exists. Our formalisation provides hints on the use of Coquelicot in other fields of mathematics than analysis, either alone or in combination with Mathematical Components.

Indeed, although Coquelicot was designed for analysis, it contains a few topological structures. Our formalisation of topological spaces and topological notions (recall Section 2.3.3 and Section 3.3.2) leads us to think that extending Coquelicot's hierarchy from below with topological spaces would make it possible to tackle general topology more thoroughly than we did (we only focused on the tools that would help us formalise the inverted pendulum).

Moreover, our choice of using vectors from Mathematical Components to represent $\mathbb{R}^n$, in continuity with Paşca's work (recall Section 3.3.1), makes it possible to reuse results from linear algebra that are already proven in Mathematical Components in order to prove theorems in analysis. Combining Coquelicot and Mathematical Components in order to mechanise proofs that involve both analysis and algebra is not a new idea [BBRS16, Ber17], but existing approaches (including ours) suffer from a certain heaviness that we will discuss in Section 4.3.3.

4.2 Possible Extensions

This case study could be extended in two different ways that actually correspond to the parts of Figure 1.8 that are out of the scope of what we achieved so far. First, we could improve this formalisation in order to have a complete proof of stability (see Section 4.2.1). Then, we could progress towards a certified implementation of the control function we proved correct (see Section 4.2.2).

4.2.1 Completing the Proof of Stability

Our proof of stability for the inverted pendulum is actually incomplete, since we admitted some hypotheses of LaSalle's invariance principle: we assume the existence and uniqueness of solutions and their continuity with respect to initial conditions. As we explained in Section 3.2.1, it is possible to prove these properties using the Cauchy-Lipschitz Theorem.

We mentioned in Section 2.4.3 that no mechanised version of this theorem compatible with the context of this case study is available and that proving this theorem would be possible using a fixed-point theorem formalised by Boldo et al. [BCF+17]. However, in front of the issues we will discuss in Section 4.3, we decided to develop a new library for analysis in Coq. Formalising the Cauchy-Lipschitz Theorem using this new library would thus require first a substantial effort on the development of the theory of integration, which is work in progress.

Applying the Cauchy-Lipschitz Theorem to our example is unfortunately not obvious: the domain of definition of the function that defines the differential equation must be an open subset of $\mathbb{R}^n$, while we work on the compact set K. It is impossible to consider an open subset of K for this proof because its interior is empty: the components at indices 2 and 3 of the points in K, corresponding to $\cos\theta$ and $\sin\theta$ (recall Section 3.1.2), describe a circle. It is thus necessary to choose another representation of the state space, where the equivalent of K will have a non-empty interior. For instance, we could directly use θ instead of its sine and cosine. The state space would then be $\mathbb{R}^4$ instead of $\mathbb{R}^5$ and the definition of K would resemble the following one:

$$K = \left\{p \in \mathbb{R}^4 \mid V(p) \leqslant k_0\right\},$$

which gives the obvious open subset of K

$$\left\{p \in \mathbb{R}^4 \mid V(p) < k_0\right\}$$

for the application of the Cauchy-Lipschitz Theorem.

This would lead to a third representation of the differential equation that models the behaviour of the inverted pendulum. The multiplicity of the representations of the differential equation also raises another point with respect to which our formalisation is incomplete: we worked on the solutions of Equation (3.3), but we did not prove that it was equivalent to Equation (3.1). Moreover, it should also be possible to prove Equation (3.1) from the laws of physics, once we admit which characteristics of the pendulum are relevant and which parts (e.g. friction or the pole's mass) should be neglected.

4.2.2 Towards a Certified Implementation

This formalisation could be used in order to obtain a certified implementation of the control function that could run on an actual robot. As explained in Section 1.3.1, we need to take into account the discretisation of time and approximations in order to prove correct a concrete implementation.

The impact of discretisation is very similar to the use of a numerical approximation scheme for solutions of a differential equation. For a stable approximation scheme, we know that the approximations get closer to the solutions when the discretisation step decreases. By analogy, we believe that if the processor and the sensor measurements have a high enough frequency, the behaviour of solutions should not deviate too much from the one of ideal exact solutions

of the differential equation. We think the same proof techniques as the one used to prove the stability of an approximation scheme could be used for this step.

We expect that taking into account approximations would be harder. Since we can only work with a model of the sensor precision and of the engine response to its inputs, we get another incertitude on the state of the system. We must also switch from axiomatised exact real numbers to floating point numbers, which introduces rounding errors. This last kind of approximation could be taken into account in a semi-automated way, for example using the Flocq library [BM11] and Gappa [BFM09, DM10]. Since this last step consists in proving that changing the data structures on which the algorithm operates does not affect its result, refinement (which we introduce in Chapter 8) might be an option too.

4.3 Remaining Complications

Although we developed tools that solve some issues we encountered while working on this case study, we could not solve all of them. The remaining issues motivated us to develop a new library, which we will describe in Part II.

We had in particular to perform proof steps that are unnatural for a mathematician (see Section 4.3.1). We also missed some tools that were either not implemented yet or restricted to a domain of application that does not fit our example (see Section 4.3.2). Finally, we discuss in Section 4.3.3 the difficulty to combine several hierarchies of mathematical structures.

4.3.1 Discrepancy with Pen-and-Paper Mathematics

Our set of notations is a step towards a better resemblance between formal proofs and their pen-and-paper equivalent. However, they are not sufficient: some proofs are unnatural because they differ too much from usual mathematics.

In particular, issues arise from the fact that we want to do classical reasoning using a library that was designed, admittedly from a classical axiomatisation of real numbers, but with an emphasis on constructive proofs. For instance, it is impossible using only Coq's standard library and intuitionist reasoning to prove that, if l is the least upper bound of the set A, then for all $\varepsilon > 0$ there exists $p \in A$ such that $l - \varepsilon \leqslant p \leqslant l$. In other terms, Coq's axiomatisation of real numbers is not expressive enough to give an arbitrary approximation of a least upper bound.

This issue is easy to solve since we can (and we did) prove this property using additional axioms allowing for classical reasoning. However, this is not the only example where this choice of design is problematic. We mentioned in Section 2.3.3 Coquelicot's definition of closed sets: a set is closed if and only if its complement is open, which is written in Coquelicot with a double negation.

```
Definition closed {U : UniformSpace} (A : set U) :=
  forall p, ~ (locally p (~‘ A)) -> A p.
```

This double negation in the definition of closed sets makes some (constructive) proofs possible, while they would have been impossible without it: removing a double negation is a classical reasoning step.

Another difference with the practice of mathematicians is the necessity, in proofs involving asymptotic reasoning, to anticipate how elements of filters will be built during the proof. To

be more precise, filters constitute an abstraction over $\varepsilon - \delta$ definitions and mathematical proofs involving these definitions often requires to split the epsilons. With pen-and-paper, mathematicians often do not bother splitting them since for instance properties of the form

$$\forall \varepsilon > 0.\ e < \varepsilon$$

and

$$\forall \varepsilon > 0.\ e < 4\varepsilon$$

are equivalent. The important point is that mathematicians do not have to know **in advance** that they will end up with 4ε instead of ε. In formal proofs however, although Coquelicot already provides tools to combine results on filters without unfolding their definition, it is usually necessary to know beforehand how the splitting of epsilons will be done to reach ε at the end of the proof, or equivalently to know which will be the multiplying factor in front of ε at the end, which is unsatisfactory.

4.3.2 Missing Tools

The absence of some tools made our proofs harder than they should. We already discussed issues with the iteration of the maximum operator in Section 3.3.1. Our solution, duplication of code from the Mathematical Components library, removing the necessity of a monoid structure and adding hypotheses is not really satisfactory and is obviously a short-term solution.

Another missing tool is a function that, given a function f and a point x, computes the differential $\mathrm{d} f_x$. There is only the ternary predicate `filterdiff` in Coquelicot to express properties on differentials, but this predicate does not give access to the function $\mathrm{d} f$.

On the opposite, Coquelicot provides the `Derive` function that takes a function from $\mathbb{R}$ to itself and returns a total function that represents its derivative function [BLM12]. This function makes proofs easier because it decorrelates the proof of derivability for a function and the proof of the property one wants to show about the derivative of this function. Moreover, the way the `Derive` function is defined sometimes removes the need for derivability hypotheses. For instance,

```
Lemma Derive_scal :
  forall f k x, Derive (fun x => k * f x) x = k * Derive f x
```

is a theorem in Coquelicot without any derivability hypothesis.

The equivalent of the `Derive` function for differentials is unfortunately not implemented in Coquelicot.

4.3.3 Combining Several Hierarchies

The main obstacle to the simultaneous use of different libraries is the lack of compatibility between the different definitions of the same mathematical object. For instance, we explained in Section 3.4.2 that Schepler's library for topology in Coq [Scha] is hard to use in combination with Coquelicot [BLM15] since the definitions of filters and uniform spaces are incompatible. We decided to prove again the theorems from Schepler's library that we needed, but using Coquelicot.

Another option, which we used to combine Coquelicot and Mathematical Components, is to provide an interface between both libraries. In our case, this was very simple since we only had to instantiate Coquelicot's hierarchy on the type of row vectors from Mathematical Components. Some more complex operations may however be required. Bernard et al. [BBRS16] for instance had to prove the correspondence between two different definitions of the factorial function, or to use coercions and morphism lemmas to translate statements on non-negative integers from the type `R` to the type `nat`.

Making use of translations from one library to the other quickly becomes extremely tedious, especially if the hierarchy of algebraic structures itself is duplicated. Both Coquelicot and Mathematical Components contain their own interfaces describing groups, rings, modules and normed spaces, and these structures are not compatible, although the axioms we will present in Section 5.1.1 make some of them equivalent.

In front of these difficulties, we decided to implement a new library for analysis in Coq. This library, named Mathematical Components Analysis [ACM+], reimplements and extends Coquelicot's tools for analysis, based on a hierarchy that is by design compatible with Mathematical Components (see Chapter 5).

Part II

Designing a Library of Mathematics

Part II
Designing a Library of Mathematics

CHAPTER 5

HIERARCHY OF THE MATHEMATICAL COMPONENTS ANALYSIS LIBRARY

Having a well-organised hierarchy of algebraic and topological structures is important for a library to be easy to use. It also must be well-tuned in order to maximise sharing. The issues mentioned in Section 4.3 partly come from the necessity to combine the Coquelicot [BLM15] and Mathematical Components [MCT] libraries. We thus decided to provide a unified framework for real analysis, Mathematical Components Analysis [ACM+], taking benefits from the algebraic theory of Mathematical Components to rework Coquelicot (see Section 5.2). We discuss in Section 5.1 the foundations of this library. We extended Coquelicot's hierarchy with structures that allow for a better integration of our contributions from Part I (see Section 5.3). Finally, we are currently reworking some interfaces in the hierarchy in order to make a better use of tools that solve other issues of the standard library (see Section 5.4).

This work was done in collaboration with Reynald Affeldt and Cyril Cohen, except for the current work described in Section 5.4. Assia Mahboubi and Pierre-Yves Strub also worked on the logical foundations of the library (described in Section 5.1.1). All code snippets come from the Mathematical Components Analysis library [ACM+], unless otherwise specified.

5.1 Principles of Design

Although the starting point of our library is Coquelicot's hierarchy, there is a crucial difference between both libraries, since we work in a different logical context (see Section 5.1.1). Still, we use the same principles of organisation (see Section 5.1.2), which also proved their value in the Mathematical Components library.

5.1.1 Logical Foundations

As opposed to Coquelicot, which is constructively built on top of a classical axiomatisation of the set of real numbers [May01], we make use of classical reasoning in our library. In particular, we use a set of axioms that allows for reasoning steps that are standard in classical mathematics.

We use three axioms from the standard library, which were proven to be compatible with Coq's logic [CDT19]: using these axioms does not introduce inconsistencies.

```
Axiom functional_extensionality_dep :
  forall (A : Type) (B : A -> Type) (f g : forall x : A, B x),
  (forall x : A, f x = g x) -> f = g.

Axiom propositional_extensionality :
  forall P Q : Prop, P <-> Q -> P = Q.

Axiom constructive_indefinite_description :
  forall (A : Type) (P : A -> Prop),
  (exists x : A, P x) -> {x : A | P x}.
```

The two first axioms generalise the ones we presented in Section 2.3.3. The first axiom is a strong version of functional extensionality, called `funext` in Section 2.3.3. This axiom generalises `funext` to dependent functions. The second axiom is the same as `propext`, but we kept the name from the standard library.

The third axiom makes it easier to deal with witnesses for existential propositions. In Coq, there are limitations in the ways one can use witnesses. Two kinds of existential propositions exist: the propositional one, denoted by `exists x, P x`, which lives in the sort `Prop`, and the dependent sum, denoted by `{x : A | P x}` which lives in the sort `Set` or `{x : A & P x}` which lives in the sort `Type`. A witness for the propositional version can only be used to build a term in the sort `Prop` (and similarly for the others). The third axiom makes it possible to transform an existential proposition in `Prop` into a dependent sum.

Thanks to this axiom, one can define a function that, given a proposition, computes a witness for this proposition if there is one: this is Hilbert's epsilon operator [Hil22]. We will discuss the role of such a function in Section 5.3.3.

This last axiom is a form of axiom of choice that makes it possible, in combination with propositional extensionality, to prove that any type inherits from the `choiceType` structure from Mathematical Components, or equivalently that any type has a choice function for decidable predicates. This is stated by the following lemma (we give the meaning of the name `mixin_of` in Section 5.1.2):

```
Lemma gen_choiceMixin (T : Type) : Choice.mixin_of T.
```

Moreover, it is also possible to prove a strong version of the law of Excluded Middle (using a constructive sum of propositions).

```
Lemma pselect (P : Prop): {P} + {~P}.
```

This lemma is established as a consequence of the standard law of Excluded Middle (i.e. expressed using the disjunction in `Prop`), which we prove as Diaconescu's theorem [Dia75] thanks to propositional extensionality and indefinite description.

Lemma `pselect` makes it possible to define in particular a projection from `Prop` to `bool`: the `asbool` function computes a boolean value that is logically equivalent to the proposition in argument. This equivalence is expressed using the `reflect` inductive we mentioned in Section 2.3.1.

```
Definition asbool (P : Prop) := if pselect P then true else false.

Lemma asboolP (P : Prop) : reflect P (asbool P).
```

As a consequence, any predicate becomes decidable, which strengthens the impact of Lemma `gen_choiceMixin`.

5.1.2 Organising the Library

The design technique that proved to be the most efficient and robust for large hierarchies of mathematical structures is the one called "packed classes" [GGMR09, Gar11]. We recall this methodology in this section.

A mathematical structure is usually composed of a carrier set and of operations on the carrier that must satisfy some properties. For instance, a ring is a set equipped with an addition and a multiplication that must have neutral elements and that must satisfy computation laws, such as distributivity:

$$\forall x.\ \forall y.\ \forall z.\ x * (y + z) = x * y + x * z.$$

It is possible to define the type of rings by bundling the operations and their properties in a record, but it should be done carefully so that this kind of definition scales to large hierarchies. Sharing between different structures is particularly important for efficiency and should be maximised. For example, a ring is also an additive group. It is better to provide a form of inheritance between rings and additive groups rather than copying the laws of additive groups in the definition of rings.

The methodology of packed classes gives a way to have efficient inheritance through the use of three different records:

— the mixin, named `mixin_of`, bundles the operations and properties of a structure, but only those that cannot be obtained through inheritance.

— the class, named `class_of`, is the record that allows for inheritance: it packs the mixin of the structure with the classes of the other structures it inherits from.

— the type, named `type`, packs the class with the carrier type to define the mathematical structure.



Remark

The names `mixin_of`, `class_of` and `type` are used for every structures in Mathematical Components. Thanks to Coq's module system, there is no clash of names: the mixin of the ring structure is called `Ring.mixin_of`, while the one of the additive group structure is called `Zmodule.mixin_of`.

Example

We describe here the ring structure (`ringType`) from the MATHEMATICAL COMPONENTS library [MCT].

The mixin of this structure only gives the laws related to the multiplication, since the laws about addition are obtained through inheritance from the additive group structure (`zmodType`). Since some of these laws also concern the addition (the distributivity rules), this mixin is parametrised by an additive group.

```
Record mixin_of (R : zmodType) : Type := Mixin {
  one : R;
  mul : R -> R -> R;
  _ : associative mul;
  _ : left_id one mul;
  _ : right_id one mul;
  _ : left_distributive mul +%R;
  _ : right_distributive mul +%R;
  _ : one != 0
}.
```

The class of the ring structure then bundles this mixin with the class of the additive group.

```
Record class_of (R : Type) : Type := Class {
  base : Zmodule.class_of R;
  mixin : mixin_of (Zmodule.Pack base)
}.
```

The `Pack` function in the above class is the constructor of the `zmodType` structure: it transforms a class of additive group into an element of the type of additive groups. The same structure also exists for rings.

```
Structure type := Pack {sort; _ : class_of sort}.
```

Thanks to coercions, it is possible to identify the ring with its carrier. More precisely, when we have `R : ringType`, we may write `x : R` to denote that `x` is an element of the carrier of the ring `R`.

In MATHEMATICAL COMPONENTS, hence in our library, the inference mechanism that allows to automatically infer the structure on a type is the one of canonical structures [Sai99, MT13]. Each structure comes with notations that make it possible to trigger unification in order to infer the structure. These notations have two roles: to ease the instantiation of the hierarchy on concrete types and to infer a structure for a given type when required.

For instance, for the `ringType` structure, there is a notation `RingType T m` that, given a type `T` and a ring mixin `m` for `T`, infers a class of additive group for `T` and packs all these elements into a `ringType` structure. This makes it possible to use only the ring mixin to define the full ring structure on a type. For example, the declaration of a ring structure for real numbers in our library is the following one.

```
Definition R_ringMixin := ...

Canonical R_ringType := Eval hnf in RingType R R_ringMixin.
```

The additive group class for `R` is automatically inferred from a previous declaration of a canonical additive group structure for `R`.

There is also a notation `[ringType of T]` that triggers the inference of an element of `ringType` whose carrier type is `T`.

The main ingredients of these notations are phantom types, which we do not detail here. The interested reader may refer to the book about MATHEMATICAL COMPONENTS by Mahboubi and Tassi [MT18].

5.2 The Starting Point: COQUELICOT

COQUELICOT already has a good infrastructure to deal with standard questions in real analysis, so we decided to keep it. Let us first present COQUELICOT's hierarchy (see Section 5.2.1) and then describe our modifications to this hierarchy to make it compatible with the MATHEMATICAL COMPONENTS library (see Section 5.2.2) and to fix minor issues thereof (see Section 5.2.3).

5.2.1 COQUELICOT's Hierarchy

For the reader's convenience, let us repeat Figure 2.6 as Figure 5.1.

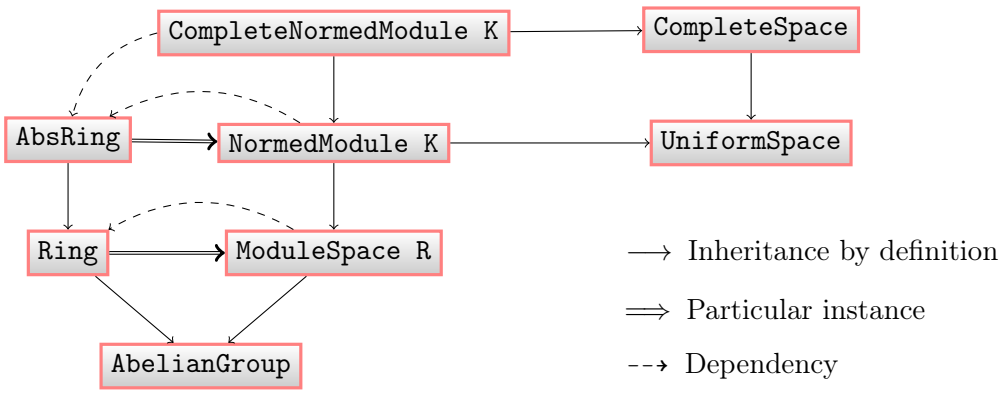


Figure 5.1 – The COQUELICOT Hierarchy (repeated)

The basic algebraic structure in COQUELICOT is the abelian group (`AbelianGroup`). It is meant to capture the additive structure, both in rings (`Ring`) and in modules over a given ring (`ModuleSpace R`). Most often, in real analysis, we deal with modules that are in fact vector spaces. The simplest example is the finite-dimensional vector space $\mathbb{R}^n$, which we used in our case study (see Section 3.3.1), but it is not the only one. Another topic of interest is the case of function spaces, such as the space $\mathcal{C}^0([a;b],E)$ of continuous functions from the segment $[a;b]$ to the vector space E, the space $\mathcal{C}_c^\infty(\mathbb{R})$ of smooth real functions with compact support or the space $\mathcal{L}^1(I)$ of real functions whose absolute value is Lebesgue-integrable on I.

Another important notion, which appears in all these examples, is the norm. Norms make it possible to define neighbourhoods and to reason about convergence, but also to develop the theories of integration and of series. Coquelicot's structures related to norms are rings equipped with an absolute value (`AbsRing`) and modules over such a ring that are equipped with a norm (`NormedModule K`).

However, norms are not a good abstraction to reason about neighbourhoods and convergence. Indeed, in a finite-dimensional vector space, all norms are equivalent (see Definition 5.1) and hence define the same notion of convergence but it is not the case in infinite-dimensional spaces. Moreover, there are spaces in which one can define a notion of convergence which does not come from a norm, e.g. metric spaces whose metric is not related to a norm. In Coquelicot, neighbourhoods are defined from the topological notion of uniform space (`UniformSpace`), which encompasses both normed and metric spaces.

Definition 5.1 (Norm equivalence)**.** Let N_1 and N_2 be two norms on a space E.

We say that N_1 and N_2 are equivalent if and only if there exist k_1 and k_2, both positive, such that for all $x \in E$

$$k_1 N_1(x) \leqslant N_2(x) \leqslant k_2 N_1(x).$$

Several equivalent definitions of the notion of uniform space exist. Let us give the one from Coquelicot here, we will discuss another one in Section 5.4.2. Coquelicot contains a variant of the pseudometric definition of uniform spaces. A pseudometric on the space E is a function $d : E * E \to \mathbb{R}^+$ which is akin to a distance function, but for which the separation condition

$$\forall x.\ \forall y.\ d(x, y) = 0 \Rightarrow x = y$$

is not necessarily true. However, in her PhD thesis [Lel15], Lelay explains that pseudometrics are not appropriate for the formalisation of the limit switching theorem (see Theorem 5.1).

Theorem 5.1 (Limit Switching)**.** *Let E_1 and E_2 be two sets and F be a complete space. Let $f : E_1 * E_2 \to F$, $e_1 \in E_1$ and $e_2 \in E_2$. Assume:*

1. *$f(x_1, \cdot)$ uniformly converges to a function $g : E_2 \to F$ when x_1 tends to e_1.*
2. *$f(\cdot, x_2)$ converges pointwise to a function $h : E_1 \to F$ when x_2 tends to e_2.*

Then, there exists l such that

$$g(x_2) \xrightarrow[x_2 \to e_2]{} l \text{ and } h(x_1) \xrightarrow[x_1 \to e_1]{} l.$$

Balls defined by a pseudometric were a more appropriate abstraction to formalise this theorem, so that in Coquelicot, a uniform space `U : UniformSpace` is a type `U` equipped with a function `ball : U -> R -> U`, where `R` is the type of real numbers from Coq's standard library, which satisfies the following properties

```
ball_center : forall (x : U) (ε : posreal), ball x ε x,
ball_sym : forall (x y : U) (ε : R), ball x ε y -> ball y ε x,
ball_triangle : forall (x y z : U) (ε1 ε2 : R),
  ball x ε1 y -> ball y ε2 z -> ball x (ε1 + ε2) z.
```

Finally, a structure which appears in Theorem 5.1 is the notion of complete uniform space (`CompleteSpace`), which also extends to normed modules (`CompleteNormedModule K`). Complete uniform spaces are a generalisation of complete metric spaces. A complete metric space is a metric space in which every Cauchy sequence converges. Similarly, a complete uniform space is a uniform space in which every Cauchy proper filter converges (recall Section 2.3.2 for the definition of (proper) filters and filter convergence, and see Definition 5.2 for Cauchy filters).



Remark

In COQUELICOT, complete spaces are also constrained by the fact that extensionally equal filters must have arbitrarily close limits, i.e. if `F` and `G` have the same elements, then for every positive real number ε, `lim F` and `lim G` are ε-close.

Using the axioms we described in Section 5.1.1, if `F` and `G` are extensionally equal, then they are equal so that `lim F = lim G`, which means that this assumption is not necessary in MATHEMATICAL COMPONENTS ANALYSIS.

Definition 5.2 (Cauchy filter)**.** A filter F is a Cauchy filter if and only if for all positive real number ε, F contains a ball of radius ε.



Remark

Definition 5.2 indeed generalises the notion of Cauchy sequence: a sequence `u` is Cauchy if and only if the filter `u @ \oo`, where `\oo` is a notation for the `eventually` filter (recall Section 2.3.2), is Cauchy.

5.2.2 Making COQUELICOT Compatible with MATHEMATICAL COMPONENTS

The biggest obstacle for a combined use of the COQUELICOT and MATHEMATICAL COMPONENTS libraries is the fact that the same mathematical structure has two different representations. This is the case of the abelian group, ring and module structures (see Table 5.1 for the correspondence between the structures from COQUELICOT and MATHEMATICAL COMPONENTS). Note however that the structures from MATHEMATICAL COMPONENTS inherit from the `eqType` and `choiceType` structures: all algebraic structures thus have a decidable equality and a choice function for decidable predicates. This is not the case of COQUELICOT's structures, except if we add the axioms discussed in Section 5.1.1.

COQUELICOT	MATHEMATICAL COMPONENTS
`AbelianGroup`	`zmodType`
`Ring`	`ringType`
`ModuleSpace R`	`lmodType R`

Table 5.1 – Corresponding Algebraic Structures in COQUELICOT and MATHEMATICAL COMPONENTS

As a consequence, we decided to remove the algebraic structures from COQUELICOT in order to use the ones from MATHEMATICAL COMPONENTS. This means in particular that the algebraic hierarchy from MATHEMATICAL COMPONENTS must be instantiated on the type of real numbers, which is possible thanks to the axioms of Section 5.1.1. We use a file called `Rstruct.v`, that contains such an instantiation, and which we believe originated from Paşca's work on Kantorovitch's Theorem [Paş08] and Newton's Method [Paş11] and was later used and extended in the COQAPPROX library [BJMD+] and by Bernard [Ber].

Another issue that makes it difficult to use both COQUELICOT and MATHEMATICAL COMPONENTS is the way norms and absolute values are handled. On one hand, in COQUELICOT, both norms and absolute values take their values in the set of real numbers. On the other hand, in MATHEMATICAL COMPONENTS, as of today, norms only come with ordered integral domains (`numDomainType` structure), or with more structure, and have their values in such a domain. Our choice in MATHEMATICAL COMPONENTS ANALYSIS was to keep the `AbsRing` structure, renamed as `absRingType`, with COQUELICOT's mixin, i.e. an absolute value with values in `R`. But we added the class of `numDomainType` to its class in order for `R`, which is both an `AbsRing` and a `numDomainType`, to have an absolute value and a norm which are definitionally equal. The `absRingType` structure will eventually be replaced with another one which is similar to `numDomainType`, but with less constraints (see the discussion in Section 5.4.1). This change will also impact the definition of norms in normed modules

This leads to the hierarchy depicted in Figure 5.2, which is not the full hierarchy of this library (see Section 5.3 for extensions of this hierarchy). The structures were renamed in order to follow the naming conventions from the MATHEMATICAL COMPONENTS library. The correspondence between the structures from COQUELICOT and the ones from MATHEMATICAL COMPONENTS ANALYSIS is given in Table 5.2.

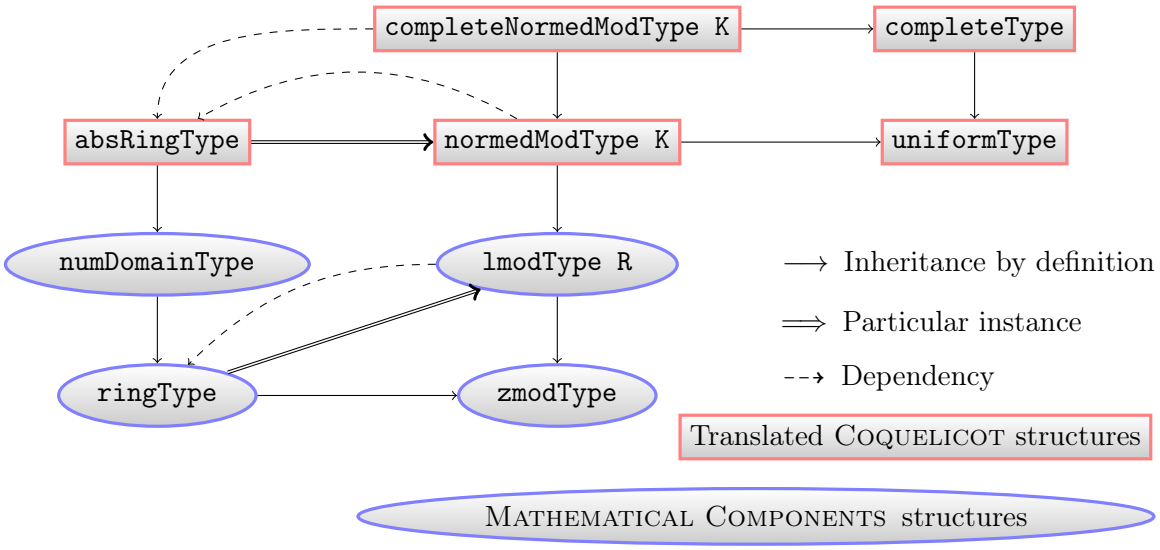


Figure 5.2 – Partial Hierarchy of MATHEMATICAL COMPONENTS ANALYSIS

COQUELICOT	MATHEMATICAL COMPONENTS ANALYSIS
`AbsRing`	`absRingType`
`UniformSpace`	`uniformType`
`CompleteSpace`	`completeType`
`NormedModule K`	`normedModType K`
`CompleteNormedModule K`	`completeNormedModType K`

Table 5.2 – Corresponding Structures in COQUELICOT and MATHEMATICAL COMPONENTS ANALYSIS

5.2.3 Minor Improvements in the Hierarchy

We disagree with the developers of COQUELICOT on two particular points in the interfaces of normed structures: the behaviour of norms with respect to multiplication, and the way the compatibility between uniform balls and norms is expressed.

Norms and multiplication

In COQUELICOT, the interface of ring with absolute value axiomatises Bourbaki's notion of semi-absolute value [Bou74]: the absolute value of a product is not necessarily equal to the product of the absolute values, but less than or equal to this product.

```
abs_mult : forall x y : K, abs (x * y) <= abs x * abs y.
```

This property also affects the interface of normed module: the norm of the product of a vector by a scalar has to be less than or equal to the product of the absolute value of the scalar and the norm of the vector.

```
norm_scal : forall (l : K) (x : V), norm (l *: x) <= abs l * norm x.
```

This had the consequence to prevent us from proving that if a linear map between two normed modules is locally bounded at the origin, then it is continuous, because we needed to use the inequality the other way around. Moreover, semi-absolute values are quite anecdotal, to the extent that they only appear in exercises in Bourbaki's book [Bou74], and make proofs harder. Indeed, instead of simply using the `rewrite` tactic, one has to combine `abs_mult` with the transitivity of inequality, which leads to unnatural proofs from the point of view of a mathematician.

Thus, we decided to replace these inequalities by equalities, using the notations `` `|x| `` for the absolute value, `` `|[x]| `` for the norm, and `x%real` for the interpretation scope of real numbers (see COQ's reference manual [CDT19] for an explanation of interpretation scopes).

```
absrM : forall x y : K, `|x * y| = `|x|%real * `|y|%real.
normmZ : forall (l : K) (x : V), `|[l *: x]| = `|l|%real * `|[x]|.
```

Remark

We tried as much as possible to follow the naming conventions and the notations from MATHEMATICAL COMPONENTS. In particular, the notation for the absolute value is the same as the one for the norm in the `numDomainType` structure, which is consistent with our will to replace `absRingType` with a structure which is closer to `numDomainType`.

Norms and balls

Finding a proper way of expressing the compatibility between norms and balls is a slightly harder issue. Lelay [Lel15] explains that the issue arises when trying to give a structure to the set $\mathbb{C}$ of complex numbers either as a $\mathbb{C}$-module or as a $\mathbb{R}$-module. $\mathbb{C}$ is naturally a ring with absolute value, the absolute value being the modulus $|a + \mathrm{i}\, b| = \sqrt{a^2 + b^2}$, and as such a normed module over itself with the norm being equal to the absolute value. This norm thus corresponds to the Euclidean norm $\|(u, v)\|_2 = \sqrt{\|u\|^2 + \|v\|^2}$ on the product $U * V$ of two normed spaces.

However, the natural uniform space structure on the product of two uniform spaces define balls in a way which corresponds to the infinity norm: the distance between two pairs is the maximum of the distance between corresponding components, or in other terms, the ball of radius ε centred on a pair p is the set of pairs whose first (respectively second) component is in the ball of radius ε centred on the first (respectively second) component of p.

```
forall (u u' : U) (v v' : V) (ε : R),
  ball (u,v) ε (u',v') = ball u ε u' ∧ ball v ε v'.
```

As a consequence, the balls defined by the norm are different from the uniform balls. Lelay hence uses the equivalence of the Euclidean and infinity norms (recall Definition 5.1) to make it possible to have balls corresponding to the infinity norm while using the Euclidean norm: she introduces in the `NormedModule` interface a coefficient, `norm_factor`, with the following two axioms:

```
norm_compat1 : forall (x y : V) (ε : R),
  norm (y - x) < ε -> ball x ε y,
norm_compat2 : forall (x y : V) (ε : posreal),
  ball x ε y -> norm (y - x) < norm_factor * ε.
```

Once again, this makes proofs unnatural, since one would expect balls in a normed space to be defined from the norm, and harder, for one has to anticipate the use of the transitivity of inequality and to put `norm_factor` beforehand in the right places. Moreover, this implies two definitions of the neighbourhood filter that are extensionally equal but not convertible (`locally` and `locally_norm` in COQUELICOT).

We favour another option, which is having uniform balls and balls defined from the norm be the same notion, and decorrelating the neighbourhood filter from the notion of ball so that one can for instance use the Euclidean norm on $\mathbb{C}$ while still having a neighbourhood filter defined from the infinity norm (see Section 5.3.1 for the link between neighbourhoods and balls). This is possible because equivalent norms define the same notion of neighbourhood.

Hence, we removed the coefficient `norm_factor` and replaced the two compatibility axioms with the following one:

```
ball_normE : ball_ norm = ball,
```

where `ball_` is the function that takes a norm function and returns the notion of ball defined from this norm.

```
Definition ball_ {V : zmodType} (norm : V -> R) (x : V) (ε : R) :=
  [set y | norm (x - y) < ε].
```

5.3 Extension of the Hierarchy

Although the hierarchy we inherited from Coquelicot (see Figure 5.2) contains the essentials for analysis, it lacks a few structures that become significant in the context of our contributions described in Part I. First, as explained in Section 3.3.2, a structure for topological spaces (see Section 5.3.1) was missing. Then, the mechanism for filter inference we presented in Section 2.3.2 was integrated into a dedicated structure which is now part of the hierarchy (see Section 5.3.2). Finally, non-empty spaces, combined with the axioms discussed in Section 5.1.1, offer a fresh perspective on limits and derivatives (see Section 5.3.3).

5.3.1 Topological Spaces

Let us briefly recall Definition 3.1: a topological space is a set equipped with a family of subsets, called open sets, which is stable by union and by finite intersection and which contains the full set and the empty set. Our first implementation of this definition, described in Section 3.3.2, stick to this definition. In the following, the structure describing topological spaces is denoted by `topologicalType`.

In a hierarchy where both topological and uniform spaces exist, we are faced with two competing definitions of neighbourhoods: a set `A : set T`, where `T` is a type equipped with a topological space structure, is a neighbourhood of a point `p : T` either because it contains an open set that contains `p`

```
locally p A = exists B : set T, open B ∧ B p ∧ B `<=` A,
```

or because it contains a ball, which is not necessarily open, that contains `p`

```
locally p A = exists ε : posreal, ball p ε `<=` A.
```

It is essential to ensure that both notions coincide for the hierarchy to be consistent, since uniform spaces are a particular case of topological spaces. In particular, in the class defining the `uniformType` structure, we put the following field for inheritance

```
base : Topological.class_of T.
```

This fact seems to imply that the way to proceed is to define neighbourhoods from open sets and then add compatibility conditions in the mixin of the `uniformType` structure to ensure that balls define the same neighbourhoods. However, we have a use for a more primitive notion of "neighbourhood" filter (see Section 5.3.2). Thus, we have to make sure that both notions coincide with this primitive `locally`. We will explain how to grant this compatibility in Section 5.3.2. Here we focus on topological spaces.

The above equality between `locally` and the open-based definition of neighbourhoods is a necessary compatibility condition in the mixin of the `topologicalType` structure, but we can go even further in the use of a primitive neighbourhood filter. Indeed, open sets can be recognised in the fact that they are neighbourhoods (in the sense of the open-based definition) of all their points. One could thus build the `open` predicate from `locally` using this property as definition. In order to prove all properties of open sets from this definition, it is however necessary to know that `locally p` is a proper filter (recall Definition 2.7) for every `p`. This leads to the following mixin for the `topologicalType` structure:

```
Record mixin_of {T : Type} (locally : T -> set (set T)) := Mixin {
  open : set (set T) ;
  ax1 : forall p : T, ProperFilter (locally p) ;
  ax2 : forall p : T, locally p =
    [set A : set T | exists B : set T, open B ∧ B p ∧ B `<=` A] ;
  ax3 : open = [set A : set T | A `<=` locally^~ A ]
}.
```

Remark

We could have inlined the `ax3` axiom in the `ax2` axiom and have the `open` predicate as a definition outside of the structure, but for some instantiations open sets actually are the primitive notion and we do not want to enforce a definition of the predicate which would not be definitionally equal to this primitive notion. This is why the predicate appears as a field of the structure.

We provide factories, i.e. functions that build structures from some arguments, in order to build topological spaces either from neighbourhoods (deducing open sets) or from a family/*base*/*subbase* (see Definition 5.3) of open sets (deducing neighbourhoods).

Definition 5.3 (Open (Sub)base)**.** A family $(O_i)_{i\in I}$ of sets defines:

— a base for a topology if and only if $\bigcup_{i\in I} O_i$ covers the whole set and

$$\forall i \in I. \forall j \in I.\ \forall p \in O_i \cap O_j.\ \exists k \in I.\ p \in O_k \text{ and } O_k \subseteq O_i \cap O_j.$$

— a subbase for a topology if and only if the collection of all finite intersections of elements of this family defines a base.

5.3.2 Filtered Spaces

Up to this point, the `locally` function could be a field of the `topologicalType` mixin instead of being more primitive. However, neighbourhoods in a topological space are not the only kind of neighbourhoods we are dealing with. Some neighbourhoods cannot be expressed as neighbourhoods in the topological sense. For instance, "neighbourhoods of $+\infty$" (recall `Rbar_locally +oo` from Section 2.3.2) are subsets of `R` while `+oo` is not a point in this topological space.

A unique `locally` function capturing all kinds of neighbourhoods can however be defined thanks to a canonical structure [Sai99, MT13]: this is the purpose of the inference mechanism

we described in Section 2.3.2. Hence, we integrated this mechanism into a new structure placed lower in the hierarchy than topological spaces, also simplifying the way it works.

We designed a family of types `T : filteredType U` such that elements of `T` are associated to sets of sets on `U` through the `locally : T -> set (set U)` function. This family exactly corresponds to the `canonical_filter` structure of the mechanism we described in Section 2.3.2. We do not enforce that `locally t` is a filter for any element `t` of `T`, since this structure is just designed for sharing purposes. This is enforced in the mixin of the `topologicalType` structure (recall `ax1` in the mixin page 82).

As in Section 2.3.2, having `T` different from `U` allows for `locally +oo` to be equal to `Rbar_locally +oo`, thanks to an instantiation of the `filteredType R` structure as the canonical filter on `R` associated to `+oo : Rbar`.

```
Canonical Rbar_filter := FilteredType R Rbar Rbar_locally.
```

Here, the `FilteredType U T loc` notation builds the structure of type `filteredType U` associated to `T`, the `locally` function being defined as the `loc` function.

It is then possible to get back the notations from Section 2.3.2 using `locally` in their definition. It is still however necessary to keep a dedicated structure to match the source type of an arrow type. This structure is the same as the one from Section 2.3.2, up to renaming. We also developed on top of the `filteredType` structure a set of notations and tactics that make the manipulation of filters smoother (see Section 6.1).

Inheritance then requires compatibility conditions in the structures describing topological and uniform spaces. We already discussed such conditions in Section 5.3.1 for topological spaces. Note that the `locally` function for a topological space `T` has type `T -> set (set T)` instead of `T -> set (set U)`. Indeed, the compatibility conditions imply that the `locally` function associates to any element of a topological space its neighbourhood filter, which means that `T` and `U` are the same. This appears in the definition of the class for the `topologicalType` structure.

```
Record class_of (T : Type) := Class {
  base : Filtered.class_of T T;
  mixin : mixin_of (Filtered.locally_op base)
}.
```

The compatibility conditions for uniform spaces must ensure that neighbourhoods correspond to the uniform topology, i.e. are defined using balls: a set `A` is a neighbourhood of a point `p` if and only if `A` contains a ball centred on `p`.

This kind of constructions, sets that contain a set from a given family, is actually a standard way of building filters (recall our example page 28). Hence, we designed a specific function that builds a filter from a family of sets.

```
Definition filter_from {I T : Type} (D : set I) (B : I -> set T) :=
  [set P | exists2 i, D i & B i `<=` P].
```

Here, `D` should be understood as the domain of indices and `B` defines the family. If the family is a filter base (see Definition 5.4), then `filter_from D B` is indeed a filter. If moreover no element of the family is empty, then it is a proper filter.

```
Lemma filter_from_filter (I T : Type) (D : set I) (B : I -> set T) :
  (exists i : I, D i) ->
  (forall i j, D i -> D j -> exists2 k, D k & B k `<=` B i `&` B j) ->
  Filter (filter_from D B).

Lemma filter_from_proper (I T : Type) (D : set I) (B : I -> set T) :
  Filter (filter_from D B) -> (forall i, D i -> B i !=set0) ->
  ProperFilter (filter_from D B).
```

Definition 5.4 (Filter Base)**.** Let $(B_i)_{i\in D}$ be a family of sets.

We say that $(B_i)_{i\in D}$ constitutes a filter base if and only if D is non-empty and any intersection of two elements of $(B_i)_{i\in D}$ contains another element of the family:

$$\forall i \in D.\ \forall j \in D.\ \exists k \in D.\ B_k \subseteq B_i \cap B_j.$$

Using this construction, it is then easy to define neighbourhoods from balls.

```
Definition locally_ {T T' : Type} (ball : T -> R -> set T') (x : T) :=
   @filter_from R _ [set x | 0 < x] (ball x).
```

The compatibility condition to put in the mixin for the `uniformType` structure is then no more than the equality between the `locally` function and the neighbourhood filter generated from balls:

```
locally = locally_ ball.
```

5.3.3 Non-Empty Spaces

An important tool for proofs in analysis is the ability to get witnesses for existential propositions. This is pervasive in the mathematical practice: whenever we know a set is non-empty, we give a name to one of its elements and manipulate it. An important difference between classical and intuitionist mathematics comes from the fact that in intuitionist logic one has to build an explicit witness for existential propositions while it is not the case in classical proofs. Thus, in constructive mathematics we always know which elements of the set we are manipulating while in classical mathematics we are satisfied with the knowledge that it exists. Axioms such as the Axiom of Choice make it possible to prove more existential propositions and thus to manipulate more witnesses.

Coq's logic imposes limitations in the ways one can use witnesses, depending on the kind of existential proposition that is used (recall Section 5.1.1). This is an issue when manipulating sets. Indeed, sets are most easily represented as predicates, i.e. functions with values in `Prop`. This means that if we know a set `A` is non-empty, i.e. the proposition `exists x, A x` holds, then the element of `A` we get from this proposition can only be used to prove other propositions and not to build for instance other points in `A`.

In Mathematical Components [MCT], it is possible to get round this issue by using predicates with values in `bool`. In types equipped with a choice function (`choiceType`), two functions are defined: `xchoose`, which takes a boolean predicate `P` and a proof that it is satisfiable (i.e. of `exists x, P x`) and returns an element of `T` that satisfies `P`, and `choose`,

which does the same but requires an element of `T` instead of a proof of `exists x, P x` and returns this default element if `P` is not satisfiable.

Using the axioms described in Section 5.1.1, any type becomes a `choiceType`. Hence, we could use the constructions from Mathematical Components to work with sets. However, propositions are more natural than boolean expressions to express properties, and in particular undecidable ones: with our set of axioms, the only way to give a boolean predicate for an undecidable property is to write the corresponding proposition and then to use the `asbool` projection from `Prop` to `bool`.

Constantly switching between `Prop` and `bool` in proofs is tedious, so we built functions similar to `xchoose` and `choose`, but for propositional predicates. The equivalent of `xchoose` is called `xget` and `get` corresponds to `choose`. There is however a slight difference between `get` and `choose`: `get` does not require a default element. Indeed, in mathematics spaces are never empty and most often have a distinguished point (e.g. 0 in rings, fields or vector spaces). Thus, we built a structure called `pointedType` for types that have a canonical inhabitant (`point`). It inherits from the `choiceType` structure and is placed at the bottom of the hierarchy in Mathematical Components Analysis, which is fully depicted in Figure 5.3.

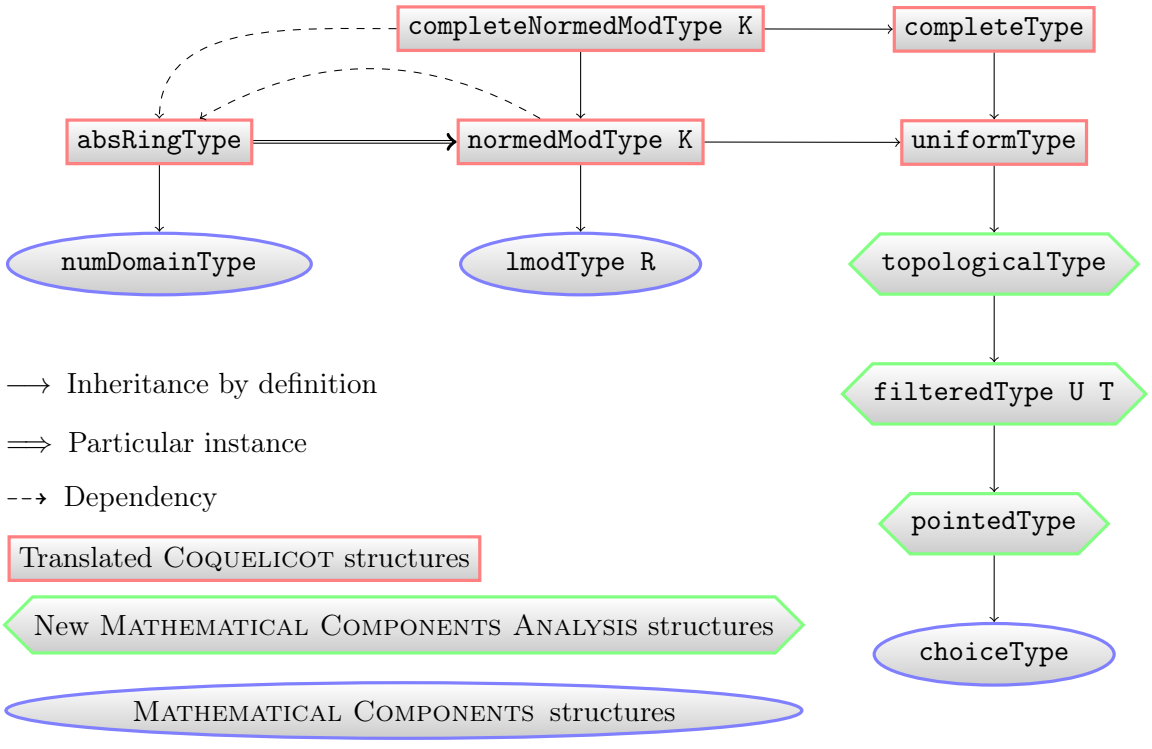


Figure 5.3 – Hierarchy of Mathematical Components Analysis

The `get` function is then defined on types that have a `pointedType` structure: `get P` is `point` if `P` is not satisfiable, and a witness for `P` otherwise. The `get` function thus plays the role of Hilbert's epsilon operator [Hil22]. This function has many applications. In particular, we used it in our proofs of Zorn's Lemma, directly inspired from the one by Schepler [Schb] [1],

1. We did not keep the dependency on Schepler's library on set theory.

and of Tychonoff's Theorem. Other applications include the theories of limits and derivatives that we will now detail.

Limits

The `get` function allows for a generalisation of the `Lim` function from the Coquelicot library [BLM15]. The `Lim` function computes the limit of a real function at any point in $\overline{\mathbb{R}}$. In fact, it can be defined from the `lim` function, which computes the limit of a filter: the limit of `f` at point `x` is the limit of the filter `f @ x`, i.e. `Lim f x = lim (f @ x)`.

In Coquelicot, the `lim` function is defined only for filters on a complete space and can be used only for Cauchy filters, since we know they converge. Removing the constraint of constructivism, and using the `get` function, it is possible to define a function computing the limit of **any filter**.

```
Definition lim_in {U : Type} {T : filteredType U} :=
  fun F : set (set U) => get (fun l : T => F --> l).
```

We provide the notations `[lim F in T]` and `lim F` to represent the limit of filter `F` in `T : filteredType U`. In the latter, the type `T` is inferred.

Of course, it is possible to prove properties on the limit of a filter only if we know it converges. This comes from the characterisation of the `get` function: `get P` is a point that satisfies `P` if we know `P` is satisfiable.

```
Lemma getPex (T : pointedType) (P : set T) : (exists x, P x) -> P (get P).
```

Thanks to the `lim_in` function, it is possible to express filter convergence, i.e. the **existence** of a limit, **without using an existential quantifier**: a filter converges if and only if it converges to its limit.

```
Notation "[ 'cvg' F 'in' T ]" := (F --> [lim F in T]).

Lemma cvg_ex (U : Type) (T : filteredType U) (F : set (set U)) :
  [cvg F in T] <-> (exists l : T, F --> l).
```

In the same way as for `lim`, we provide the notation `cvg F`, which triggers the inference of `T` in order to build the term `[cvg F in T]`.

Derivatives

Moreover, we can also compute the directional derivative of a function from a normed module over $\mathbb{R}$ to another one, at any point, in any direction, using the `lim` function: the directional derivative of f at point a in the direction given by the vector v is

$$\lim_{\substack{h \to 0 \\ h \neq 0}} \frac{f(a + hv) - f(a)}{h}.$$

In Coq, this gives

```
Definition derive {V W : normedModType R} (f : V -> W) a v :=
  lim ((fun h => h^-1 *: ((f \o shift a) (h *: v) - f a)) @ locally' 0).
```

We denote by `'D_v f a` this directional derivative. On the particular case of real functions, a similar definition (removing the multiplication by `v`) gives the derivative function, denoted by `f^‘()` for the first derivative and `f^‘(n)` for the $\texttt{n}^{\texttt{th}}$ one. Thus, our `derive` function generalises the `Derive` function from Coquelicot.

We also explain in Section 6.2.2 how the `get` function can be used to define differentials.

5.4 Modification of the Interfaces

Originally, the Mathematical Components Analysis library contained a compatibility layer to make it possible to use proofs developed with Coquelicot without modification. We later realised that our library deviated so much from Coquelicot that keeping compatibility was a burden which was no longer meaningful.

Our library still inherits from Coquelicot a great portion of its content, parts of which we are currently modifying. In particular, normed spaces will benefit from a refactoring of the hierarchy in Mathematical Components (see Section 5.4.1). We are also currently experimenting a more abstract definition of uniform spaces (see Section 5.4.2). This new definition opens the door to a removal of the dependency of Mathematical Components Analysis on Coq's standard library (see Section 5.4.3).

5.4.1 Refactoring Normed Spaces

As we explained in Section 5.2.2, the Mathematical Components library contains structures for normed spaces which are at least an ordered integral domain. In these structures, the norm's codomain is the normed space itself, while the norm we inherited from Coquelicot has values in the set `R` of real numbers.

Our solution to keep compatibility was to make sure that the norm on `R` does not depend on the structure from which it comes: either from the normed structures of Mathematical Components or from ours. We thus imposed the inheritance from the `numDomainType` structure in our `absRingType` structure. However, this solution does not work for complex numbers since the `numDomainType` structure forces the norm to have values in $\mathbb{C}$ while the `absRingType` structure asks for a norm with values in $\mathbb{R}$.

Kazuhiko Sakaguchi is currently refactoring the hierarchy of Mathematical Components in order to generalise the structures involving norms. This new interface will provide a common base for the norms from the `numDomainType` and `normedModType` structures and for the absolute value from the `absRingType` structure. This will allow for a simplification of the `absRingType` and `normedModType` structures. In particular, the `absRingType` structure will be replaced with a new structure in Mathematical Components that combines the `ringType` structure and the new structure for normed spaces.

Similarly, the `normedModType` structure will inherit from this new structure of normed spaces. By definition, the norm on the module and the absolute value on the underlying ring will have the same codomain, which will not necessarily be `R`: an ordered ring is sufficient to state all the axioms of normed modules and rings with absolute values.

5.4.2 A More Abstact Definition of Uniform Spaces

Uniform spaces are currently defined through a reformulation of the definition based on pseudometrics (recall Section 5.2.1). Similarly as for norms, this definition involves the type `R` of real numbers. We are planning to remove the dependency on this type. We thus decided to experiment with a more abstract definition of uniform spaces that does not require real numbers.

This definition is based on the notion of *neighbourhood system*, also known as collection of *entourages* (see Definition 5.5): a uniform space is a type equipped with a collection of entourages. This definition is the one used in textbooks on topology [Bou71, Wil08].

Definition 5.5 (Entourages)**.** A neighbourhood system on a set X is a set E of subsets of $X \times X$, called entourages, such that:

— E is a filter.

— every entourage contains the diagonal set $\Delta = \{(x, x) \mid x \in X\}$.

— for all $A \in E$, the set $A^{-1} = \{(x, y) \mid (y, x) \in A\}$ is also an entourage.

— for all $A \in E$, there is an entourage $B \in E$ such that A contains the set

$$B \circ B = \{(x, y) \mid \exists z.\ (x, z) \in B \text{ and } (z, y) \in B\}\,.$$

A good intuition on entourages is to consider them as neighbourhoods of Δ. In particular, an entourage A defines a relation of closeness between points:

— a point is always close to itself: $\Delta \subseteq A$.

— if x is close to y, then y is close to x: A^{-1} is also an entourage.

The last property of entourages means that there are degrees of closeness: x and y are close to each other if we can find some z which is twice as close both to x and y. This property is a bit more restrictive as the triangular inequality for balls, where it is not necessary to use the same bound on the distance for both x and y.

This intuition naturally leads to the definition of neighbourhoods. Indeed, given an entourage A and a point x, the set of points that are close to x (with respect to A) is the set

$$A[x] = \{y \mid (x, y) \in A\}\,.$$

A neighbourhood of x is a set that contains all the points that are close to x for a certain notion of closeness: N is a neighbourhood of x if and only if $A[x] \subseteq N$ for some entourage A.

Similarly, uniform continuity has a very natural definition in terms of entourages. The $\varepsilon - \delta$ definition of uniform continuity is the following one: f is uniformly continuous if and only if

$$\forall \varepsilon > 0.\ \exists \delta > 0.\ \forall x, y \in X.\ \mathrm{d}(x, y) < \delta \Rightarrow \mathrm{d}(f(x), f(y)) < \varepsilon.$$

This definition means that $f(x)$ and $f(y)$ can be arbitrarily close to each other as soon as x and y are sufficiently close to each other in a way which does not depend on x and y but only on how close to each other we want $f(x)$ and $f(y)$ to be. In terms of entourages, this means that the function $(x, y) \mapsto (f(x), f(y))$ maps neighbourhoods of Δ to neighbourhoods of Δ. In Coq, this is written as follows.

```
Definition unif_cont {U V : uniformType} (f : U -> V) :=
  (fun xy => (f xy.1, f xy.2)) @ entourage --> entourage,
```

where `entourage` is the set of entourages of a uniform space (given as implicit argument).

We replaced the interface for uniform spaces with one based on Definition 5.5 in a branch of the repository of the Mathematical Components Analysis library which is still in experimentation. It seems to be equivalent to the definition based on balls in terms of user experience. The only flaw we noticed up to now is the lack of practicality of the last point of Definition 5.5 compared to the triangular inequality of balls.

5.4.3 Removing the Dependency on the Standard Library

Using entourages instead of balls is a first step that makes it possible to remove the dependency on the type `R` from the standard library. Indeed, we started replacing `R` with a structure developed by Assia Mahboubi and Pierre-Yves Strub, called `realType`, now integrated into the library.

The `realType` structure is defined as a combination of the `archiFieldType` and `rcfType` structures from Mathematical Components, equipped with a supremum function. A type thus represents the set of real numbers if it is an archimedean field that is real closed (i.e. the Intermediate Value Theorem holds on polynomials), in which the least upper bound property holds. We reproduce in particular the mixin of this structure here:

```
Record mixin_of {R : archiFieldType} : Type := Mixin {
  sup : pred R -> R;
   _ : forall E : pred R, has_sup E -> sup E \in ub E;
   _ : forall (E : pred R) (eps : R),
      has_sup E -> 0 < eps -> exists2 e : R, E e & (sup E - eps) < e;
   _ : forall E : pred R, ~ has_sup E -> sup E = 0
}.
```

This interface has two main benefits. On one hand, instead of instantiating the hierarchy of Mathematical Components on the type `R` of real numbers using axioms (recall our reference to the `Rstruct.v` file in Section 5.2.2), we have a structure which is already compatible with this hierarchy. On the other hand, the supremum function defined by this interface already has all desirable properties, in particular the second one in the above mixin, which we had to prove using additional axioms for the standard library (recall Section 4.3.1).

Assia Mahboubi and Pierre-Yves Strub also developed a type of "extended real numbers", that extends any `realType` with the infinities. We are currently replacing the `Rbar` type from Coquelicot with this type.

Since the `realType` structure is only an interface, a good sanity check is to verify that one can actually instantiate it with one's own formalisation of real numbers. Our version of the `Rstruct.v` file contains such an instantiation for the type `R` from the standard library. Pierre-Yves Strub is also currently formalising in the library a model of real numbers called Eudoxus real numbers: the set of real numbers is defined as a quotient on the set of "almost homomorphisms" from the additive group $\mathbb{Z}$ to itself [Art04]. Another possible future work is to develop a model of real numbers based on complete uniform spaces [Bou71]: the set of real numbers is defined as the completion of the uniform space of rational numbers, i.e. the

set of all minimal (for inclusion) Cauchy filters on the set of rational numbers [Wei16]. This motivated us to develop a formalisation of uniform spaces that does not rely on real numbers.

Part II
Designing a Library of Mathematics

CHAPTER 6

TOOLS FOR ASYMPTOTIC REASONING

We mentioned in Section 4.3.1 a discrepancy between formal and pen-and-paper proofs in asymptotic reasoning. Let us develop a bit more this argument here. One very early and trivial example when such reasoning occurs is to prove that the sum of two converging functions is converging. Indeed from

$$\begin{cases} \forall \varepsilon > 0.\ \exists \delta_f > 0.\ \forall x.\ |x-a| < \delta_f \Rightarrow |f(x) - l_f| < \varepsilon \\ \forall \varepsilon > 0.\ \exists \delta_g > 0.\ \forall x.\ |x-a| < \delta_g \Rightarrow |g(x) - l_g| < \varepsilon \end{cases},$$

we get

$$\forall \varepsilon > 0.\ \exists \delta > 0.\ \forall x.\ |x-a| < \delta \Rightarrow |f(x) + g(x) - (l_f + l_g)| < \varepsilon.$$

Formally proving this requires to show the existence of such a δ, here it may be the minimum of the two δ_f, δ_g we can get from the hypotheses applied to $\frac{\varepsilon}{2}$. Giving δ explicitly makes the proof less *stable* and less readable than it would be with a "correct" informal reasoning. By stable proof, we mean that changes in its statement, or in statements it depends on, will break only the parts of the proof where the changes actually matter. When we provide an existential witness way before using it, the distance between the place it is used (and breaks), and the place where it is introduced, makes it difficult to maintain the proof script. Indeed, the maintainer has to go back and forth in the proof script to understand how changing the witness leads to breakage.

Filters slightly improve stability and readability by hiding arithmetic reasoning. However, the explicit existential quantifiers are still replaced with forward reasoning with statements that depend on how the proof will be led (recall our discussion on splitting epsilons). Our first contribution is to solve this problem by giving a set of tactics and lemmas to handle existential variables in a consistent way (see Section 6.1).

Another common tool in informal classical analysis is asymptotic developments using Bachmann-Landau notations, often called little-o and big-$\mathcal{O}$ notations [Bac94, Lan09]. They are used to write developments such as

$$f(x) = a_0 + a_1 x + \ldots + a_n x^n + \underset{x\to 0}{o}(x^n)$$

or in the definition of the differential of a function f at point x: it is the continuous linear operator $\mathrm{d}\, f_x$ such that

$$f(x+h) = f(x) + \mathrm{d}\, f_x(h) + o(h)\,.$$

Using Bachmann-Landau notations, one performs arithmetic operations with developments and uses laws like

$$\underset{x\to 0}{o}(x^n) + \underset{x\to 0}{o}(x^n) = \underset{x\to 0}{o}(x^n)\,.$$

At first sight, the abuse of notation seems to make such a law impossible to represent as an equation on functions in a formal logic. Our second contribution is to provide a solution for this problem with a set of notations and lemmas which make the users believe that they are doing arithmetic with little-o and big-$\mathcal{O}$ at the same time (see Section 6.2).

This chapter is based on our publication on the topic [ACR18]. This work was done in collaboration with Reynald Affeldt and Cyril Cohen. All code snippets come from the Mathematical Components Analysis library [ACM+], unless otherwise specified.

6.1 Small-Scale Filter Elimination

Although filters are a good way to hide $\varepsilon - \delta$ in statements, in order to prove `F P` for some ultimately true proposition `P`, one might be tempted to replace the filter `F` with its definition. This may result in a breakage of abstraction and lead to longer and less stable proof scripts (e.g. if the filter slightly changes).

Libraries such as Coquelicot already provide tools to combine results on filters without doing any unfolding but these tools still require anticipation from the user through forward reasoning. We extend them with tactics, described in Section 6.1.1, that alleviate the user's burden. We then illustrate our tactics on an example in Section 6.1.2, which we compare with the same example in Coquelicot.

6.1.1 The `near` Tactics

The goal is to make it as easy as possible to prove that a set belongs to a filter. Let us first give an intuition of how to do such a proof before describing our set of tactics that are meant to help in this process.

Proving Filter Membership

The properties of filters (recall Definition 2.7) entail the following facts:

```
Lemma filter_app (T : Type) (F : set (set T)) : Filter F ->
  forall H G : set T, F (fun x => H x -> G x) -> F H -> F G.
```

```
Lemma filterE (T : Type) (F : set (set T)) : Filter F ->
  forall G : set T, (forall x, G x) -> F G.
```

The first lemma can be used to combine hypotheses of the form `F` H_i and a conclusion `F G` into `F (fun x =>` H_1 `x -> ... ->` H_n `x -> G x)`, and the second lemma removes the filter so that we shall prove instead the simpler goal `forall x,` H_1 `x -> ... ->` H_n `x -> G x`.

We call this technique *filter elimination*: the basic principle of filter elimination is to make the users believe that instead of proving `F G` they should instead prove `G x` directly, where it is possible to make a few assumptions about `x`. These assumptions may be arbitrarily precise as long as they are compatible with `F`: `x` may be chosen in any set belonging to `F`.

However this forces forward reasoning, since the user has to anticipate every fact H_i `x` that will be used in the proof of `G x` beforehand. This means the statements H_i have to be explicitly written by the user, and they often depend on the choice of splitting of epsilons in the rest of the proof, which was also the main source of instability of proof scripts without using filters. This clearly appears in the proofs of the lemmas of the limit switching theorem (recall Theorem 5.1) in the Coquelicot library (Lemma `filterlim_switch_1` and Lemma `filterlim_switch_2`).

We now show a novel method which frees the user from explicitly providing the statements H_i.

Description of the `near` Tactics

In order to allow for a delayed choice of the assumptions H_i, we use a record, `in_filter F`, that describes the type of the sets belonging to a filter `F`.

```
Record in_filter {T : Type} (F : set (set T)) := InFilter {
  prop_in_filter_proj : T -> Prop;
  prop_in_filterP_proj : F prop_in_filter_proj
}.
```

Lemma `filter_near_of` combines Lemma `filter_app` and Lemma `filterE` using this record to formally describe filter elimination.

```
Lemma filter_near_of (T : Type) (F : set (set T)) (H : in_filter F)
  (G : set T) :
  Filter F -> (forall x, prop_in_filter_proj H x -> G x) -> F G.
```

From now on, we use a new notation for filter membership: `\forall x \near F, G x` denotes `F (fun x => G x)` and should be read "for all `x` which is near `F`, `G x` holds". We will use this phrasing instead of the too specific "`G` is ultimately (respectively eventually) true". We also define the notation `x \is_near F` for `prop_in_filter_proj H x` for some `H` of type `in_filter F`. This notation deliberately hides `H` since it is not meant to be given by the user but rather instantiated trough the use of the `near` tactics, which we now describe.

— The `near=> x` tactic performs an "introduction".

On a goal of the form `\forall x \near F, G x`, it puts into the local context a variable `x` and a hypothesis `x \is_near F` and yields the goal `G x`. The hypothesis hides an existential variable `?H` for the neighbourhood to which `x` belongs, so that the membership proof `F ?H` is actually delayed. This is in fact a simple application of Lemma `filter_near_of`.

```
Tactic Notation "near=>" ident(x) := apply: filter_near_of => x ?.
```

We call the `x` which is now in the local context a *near-variable*. A near-variable could be defined as a variable `x` which comes with an hypothesis `x \is_near F`, hiding an existential variable, in the local context.

— The `near: x` tactic "discharges" the near-variable `x`.

On a goal of the form H_i `x` such that the hypothesis `x \is_near F` is in the context, it yields the new goal `\forall x \near F,` H_i `x`. This partially instantiates the existential variable `?H` associated with the hypothesis `x \is_near F` as the intersection of H_i and a fresh existential variable `?H'`. The user is invited to prove the goal right away.

If H_i had already been added to the set hidden in the hypothesis `x \is_near F` through a previous use of `near: x`, then the goal is immediately closed without modifying `?H`.

— The `end_near` tactic instantiates existential variables with the total set.

Once the main goal has been proven, there indeed remain existential variables that have not been instantiated. These correspond to the `?H'` in the last calls of the `near: x` tactic. They can be instantiated with the total set, since it belongs to any filter.

— Similarly as for the `near=> x` tactic, the `near F => x` tactic introduces `x` as a near-variable along with the `x \is_near F` hypothesis, once again hiding an existential variable.

However, the goal does not have to be of the form `\forall x \near F, G x` since the tactic does not act on it: it only introduces a point `x` which may be used as a temporary tool in any proof. Hence, once the assumptions on `x` are inferred, we must grant that it is possible to satisfy them, since `x` is not bound by a universal quantifier. That is why this tactic requires the filter `F` to be proper, i.e. no set `H` in `F` is empty.

After using `near F => x`, one may use `near: x` and `end_near` in exactly the same way as before.

The `near=> x` and `near F => y` tactics may be combined any number of times, and in any order. Near-variables can be discharged by using `near: z` provided that the statement contains only variables introduced before `z` was. This limitation, guaranteed by Coq's type checker, is legitimate as we must not be able to introduce circular dependencies in the existential variables.

We also provide versions of the `near=> x` and `near F => x` tactics for two variables. They may be used when the filter is defined on a product space $T*U$, as a product filter: the product of filters F and G is the filter defined from the following filter base (recall Definition 5.4):

$$(\{(x, y) \mid x \in P \text{ and } y \in Q\})_{P \in F, Q \in G}\,.$$

We provide the notation `\forall x \near F & y \near G, P x y` to denote that `P` belongs to the product filter of `F` and `G`. On such a goal, the `near=> x y` tactic introduces two near-variables and yields the goal `P x y`. If `F` and `G` are identical, we also provide the notation `\forall x & y \near F, P x y`. These binary notations could be generalised to handle n-ary products but in practice we manipulate filters on the space of vectors of size n instead, hence we may use the notation with only one variable.

6.1.2 Example: a Short Completeness Proof

We detail a proof that the type of functions from an arbitrary (choice) type to a complete type is again complete. This proof is interesting for several reasons. First, it illustrates our main technical contributions: it uses all of our tactics and demonstrates our use of filters, in particular, this proof uses two filters on two different types. Second, it shortens the original proof in Coquelicot (Lemma `complete_cauchy_fct`) from about 40 lines to 7 lines (we will briefly explain the differences between the two proofs), by removing in particular the three explicit witnesses. Finally, it shows how our work leads to formal proofs that look like informal ones: arguments can be stated without being cluttered by technical constructions of witnesses (see line 5 in the proof), the latter being delayed and constructed by resorting to lemma applications (see lines 5–6), which makes for shorter and more stable proof scripts.

Explanation of the Proof

Recall that a complete space is a uniform space in which every proper filter which is also Cauchy converges (Cauchy filters are introduced in Definition 5.2). For this particular structure, the mixin is not a record as in Section 5.1.2 since it is reduced to only one property.

```
Definition mixin_of (T : uniformType) :=
  forall (F : set (set T)), ProperFilter F -> cauchy F -> cvg F.
```

For the user's convenience, we provide another name to this property which is more convenient than `Complete.mixin_of`. We do this through Lemma `complete_cauchy`, whose proof is trivial (it suffices to unpack the `completeType` structure).

```
Lemma complete_cauchy (T : completeType) (F : set (set T))
  (FF : ProperFilter F) : cauchy F -> cvg F.
```

Our goal is to prove that this property holds for the function type `T -> U`, where `U` is complete. Formally:

```
Lemma fun_complete (T : choiceType) (U : completeType)
 (F : set (set (T -> U))) (FF : ProperFilter F) : cauchy F -> cvg F.
```

Our methodology requires that some notions are phrased in a particular way. The most direct formalisation of Definition 5.2 is the following one:

```
Definition cauchy_ex {T : uniformType} (F : set (set T)) :=
  forall e : R, 0 < e -> exists x, F (ball x e).
```

However, in order to use the `near` tactics, it is easier to use the following equivalent definition:

```
Definition cauchy {T : uniformType} (F : set (set T)) :=
  forall e, e > 0 -> \forall x & y \near F, ball x e y.
```

Indeed, the existential quantification is then encapsulated in the `near` notation and can thus be treated in a systematic way in our proofs. This rephrasing could be disturbing for users that might not immediately see these are the same concepts. Hopefully they can be convinced by simple equivalence lemmas.

```
Lemma cauchyP (T : uniformType) (F : set (set T)) :
  ProperFilter F -> cauchy F <-> cauchy_ex F.
```

Although this particular equivalence is only true for proper filters, it is sufficient since in practice we only manipulate filters that are proper.

Before describing the proof of Lemma `fun_complete`, observe that the implicit type of `cauchy` in its statement is not `T -> U` as it may appear at first sight; it is actually inferred to be `fct_uniformType T U`, the type of the functional metric space, which is a `uniformType`, as required by the definition of Cauchy filters.

We start the proof by introducing the hypothesis `Fc` that states that `F` is Cauchy (see line 1). Then, we prove an intermediate property: for all `t` of type `T`, the filter $\{\{f(\texttt{t}) \mid f \in A\} \mid A \in \texttt{F}\}$ is Cauchy in `U`. This filter can be expressed succinctly as soon as one observes that it is the image of the filter `F` by the function `fun f => f t`. More precisely, it can be written `((fun f => f t) @ F)` using the infix notation `@` that denotes the image of a filter (recall Section 2.3.2). Using a notation from Mathematical Components, this can be further abbreviated as `(@^~t @ F)` [1]. The statement to prove is thus

```
forall t, cauchy (@^~t @ F).
```

Note that we do not have to write explicitly the quantification on `t`. We may state the intermediate property with a hole (we write `cauchy (@^~_ @ F)`) which is then abstracted by Coq.

Line 2 proves this fact. The proof is very simple since it is a direct consequence of `Fc` through the fact that filters are upward closed (Lemma `filterS`). Lemma `near_simpl` used in this proof performs elementary simplifications in the `near` notations.

Line 1 also transforms this intermediate property into

```
forall t : T, cvg ((fun f => f t) @ F)
```

thanks to Lemma `complete_cauchy` and stores this hypothesis as `Ft_cvg` in the local context. In order to use Lemma `complete_cauchy` we have to remove the universal quantification, which we have to put back afterwards to obtain `Ft_cvg`. We do this by instantiating `t` with a hole (through `/(_ _)`) which will be abstracted after the application of Lemma `complete_cauchy`.

```
1 move=> Fc; have /(_ _) /complete_cauchy Ft_cvg : cauchy (@^~_ @ F).
2   by move=> t e ?; rewrite near_simpl; apply: filterS (Fc _ _).
```

At this stage, we have to prove `cvg F`, knowing that `cauchy F` holds as well as

```
forall t : T, cvg ((fun f => f t) @ F).
```

Under these hypotheses, the function `fun t => lim ((fun f => f t) @ F)` is the pointwise limit of the filter `F`. We now prove that this limit is uniform (recall Lemma `cvg_ex` from Section 5.3.3).

```
3 apply/cvg_ex; exists (fun t => lim (@^~t @ F)).
```

Under the same hypotheses as before, we now have to prove

1. We display the actual proof script with such an abbreviation but the explanation that follows keeps the more explicit `fun f => f t`.

```
F --> (fun t : T => lim ((fun f => f t) @ F)).
```

Since the right-hand side is a point of the uniform space `T -> U`, it is interpreted through our notation as the filter of neighbourhoods of this point. So it suffices to prove, for all `e` such that `e > 0`, that we have

```
\forall f \near F, ball (fun t : T => lim ((fun f => f t) @ F)) e f.
```

This goal transformation is achieved at the beginning of line 4 by the application of the Lemma `flim_ballPpos`. After application of this lemma, we are in a position to use the `near` tactics as we will explain shortly.

```
4 apply/flim_ballPpos => e; near=> f => t; near F => g => /=.
```

The first `near` tactic used at line 4 has the following consequence: we are asked to prove for all `f` which is near `F` and for all `t` that

```
ball (lim ((fun f => f t) @ F)) e (f t)
```

holds. The proof goes by introducing a function `g`, which is near `F` as well; this is also achieved at line 4 by the second `near` tactic.

We then split the ball around `g t` (using Lemma `ball_splitl`, see line 5) and are left to prove two goals:

```
ball (lim ((fun f => f t) @ F)) (e / 2) (g t)
```

and

```
ball (f t) (e / 2) (g t),
```

which will both be true when `g` is near `F`, so that we will use the `near: g` tactic. We claim that this reasoning is an informal one in the sense that this proof step does not need to be interleaved with technical proofs that can be handled later as mere consequences of `near` facts.

The first goal can be proven using `Ft_cvg` and the fact that balls are neighbourhoods of their center (Lemma `locally_ball`). The terseness of SSReflect tactics actually allows us to prove it on the same line of proof script as the application of Lemma `ball_splitl`:

```
5 apply: (@ball_splitl _ (g t)); first by near: g; exact/Ft_cvg/locally_ball.
```

The second goal can be proven for all values of `t`, when `f` is near `F`, so that we first generalise `t`, using `move: (t)`, and then discharge `g` and `f` using `near: _` twice:

```
6 by move: (t); near: g; near: f; apply: nearP_dep; apply: filterS (Fc _ _).
```

After calling `near: f`, we have to prove

```
\forall f \near F, \forall g \near F,
  forall t, ball (f t) (e / 2) (g t).
```

This is achieved by using Lemma `nearP_dep` (see line 6), which folds the two `near` notations into a single one. The goal becomes:

```
\forall f & g \near F, forall t, ball (f t) (e / 2) (g t),
```

or equivalently

```
\forall f & g \near F, ball f (e / 2) g.
```

We can conclude because the filter `F` is Cauchy and `e / 2` is obviously positive (this is automatically proven by using another set of canonical structures).

The last line of the proof script is for automatically instantiating the remaining trivial existential variables:

```
7 Grab Existential Variables. all: end_near. Qed.
```

Differences with the Proof in Coquelicot

We give here the statement of Lemma `complete_cauchy_fct`, Coquelicot's equivalent of Lemma `fun_complete`.

```
Lemma complete_cauchy_fct (T : Type) (U : CompleteSpace) :
  forall (F : ((T -> U) -> Prop) -> Prop), ProperFilter F ->
  (forall ε : posreal, exists f : T -> U, F (ball f ε)) ->
  forall ε : posreal, F (ball (lim_fct F) ε).
```

Several differences occur between these lemmas. First, let us compare the statements. Both lemmas state that for any proper filter on the function space `T -> U`, if it is Cauchy, then it converges, but both the definitions of Cauchy filters and filter convergence differ. For Cauchy filters, we use the `cauchy` predicate while `cauchy_ex` is used in Coquelicot. This forces the authors of Coquelicot to use in their proof an additional theorem, Lemma `cauchy_distance`, which states the equivalence between `cauchy_ex` and a predicate which is very close to `cauchy` once the `near` notation is unfolded.

For filter convergence, the authors of Coquelicot use an $\varepsilon - \delta$ phrasing which amounts to:

```
\forall e, 0 < e -> F (ball (lim F) e).
```

We choose instead the `cvg F` notation, which is equivalent to `F --> lim F`, i.e. `F` contains the neighbourhood filter of `lim F`. In our definition, balls are thus abstracted through the use of the topology. However, our proof of Lemma `fun_complete` still resorts to balls so we have to use an additional theorem, Lemma `flim_ballPpos` in order to get back the $\varepsilon - \delta$ phrasing.

These two differences taken into account, the two proofs follow the same reasoning. However, in Coquelicot, the reasoning steps are not in the same order since it is necessary to anticipate the two goals

```
ball (lim ((fun f => f t) @ F)) (e / 2) (g t),
```

and

```
ball (f t) (e / 2) (g t).
```

The first step after proving that the function `fun t => lim ((fun f => f t) @ F)` is the pointwise limit of the filter `F` is to introduce `e` (as we did) and to use the fact that `F` is a Cauchy filter, with the `cauchy_ex` definition, on `e / 2`, to get an element `P` of `F` such that any two elements of `P` are separated by a distance of at most `e / 2`. There is no apparent reason to do this now, since the goal is to prove

```
F (ball (fun t => lim ((fun f => f t) @ F)) e),
```

but this anticipates the remainder of the proof.

The next step is to use the fact that filters are upward closed to replace this goal with the inclusion

```
P `<=` ball (fun t => lim ((fun f => f t) @ F)) e.
```

This makes it possible to introduce the function we called `f` together with the assumption that it belongs to `P`. Note that in this proof `P` cannot be changed any more, while it was not chosen yet in ours. This explains why it was necessary to split `e` beforehand. Introducing `f`, the goal is to prove that for all `t`,

```
ball (lim ((fun f => f t) @ F)) e (f t).
```

Since we already know that any element of `P` is at distance at most `e / 2` from `f`, the goal is, as in our proof, to build a function `g` which is in `P` and which is at distance at most `e / 2` from `lim ((fun f => f t) @ F)`. Such a function exists since both `P` and the ball centred on `lim ((fun f => f t) @ F)` and of radius `e / 2` belong to the proper filter `F`, hence their intersection belongs to `F` too, and proper filters only have non-empty elements.

It is however necessary to prove first that the intersection belongs to `F` in order to get `g`, and only then to finish the proof of the main goal. In comparison, we introduce a function `g` which we know will belong to an element of `F`, but it will only be determined from the use we make of `g`. Thus, we can prove the main goal without being worried about easy but tedious proofs of filter membership.

Apart from the impact of the `near` tactics, the remaining differences in the proofs are due to naming conventions and to slight reformulation of some lemmas.

Overall, the `near` tactics allow for a better focus on the main (informal) argument and free the user from anticipating every technical fact that will make this argument correct.

6.2 Bachmann-Landau Notations

When Donald Knuth addresses the editor of the Notices of the American Mathematical Society about teaching calculus, he insists on using the big-$\mathcal{O}$ notation so that it blends smoothly into equational reasoning [Knu98]. "[I]t significantly simplifies calculations because it allows us to be sloppy but in a satisfactorily controlled way." He goes as far as "dream[ing] of writing a calculus text entitled $\mathcal{O}$ Calculus".

This section synthesises the key ideas that mechanise Knuth's dream in a provably correct fashion. We explain the principles of our mechanisation in Section 6.2.1 and illustrate it through examples in Section 6.2.2.

6.2.1 Mechanisation of Equational Bachmann-Landau Notations

The little-o and big-$\mathcal{O}$ notations are traditionally defined by

$$\begin{aligned} f = o_0(e) \text{ or } f(x) = \underset{x\to 0}{o}(e(x)) &\quad\Leftrightarrow\quad \forall \varepsilon > 0.\ \exists \delta > 0.\ \forall x.\ |x| < \delta \Rightarrow |f(x)| \leqslant \varepsilon |e(x)|, \\ f = \mathcal{O}_0(e) \text{ or } f(x) = \underset{x\to 0}{\mathcal{O}}(e(x)) &\quad\Leftrightarrow\quad \exists k > 0.\ \exists \delta > 0.\ \forall x.\ |x| < \delta \Rightarrow |f(x)| \leqslant k |e(x)|. \end{aligned}$$

For the sake of readability we gave the definitions of these notions at a neighbourhood of 0, but they are generalised to any filter in our library.

The "equality" in the notation $f = o(e)$ is a well-known abuse of notation. Indeed it is neither symmetric, since one cannot write $o(e) = f$, nor transitive, since $f = o(e)$ and $g = o(e)$ do not imply $f = g$ and not even $f \sim g$ (see Section 6.2.2 for the definition of asymptotic equivalence).

In fact, $f = o(e)$ should be read as "f **is a** little-o of e". It is not rare to see this reading enforced by the notation "$f \in o(e)$" in undergraduate-level teaching, allegedly to prevent students' confusion (see for example in a textbook from the eighties still popular in France [AF88]). It is therefore no surprise to find $o_0(e)$ viewed as a set of functions, or equivalently a predicate on functions, even in recent formalisations [GCP18].

Our formalisation builds on the set-theoretic notation using a type-theoretic variant. Indeed, we provide both a predicate `littleo_def` for functions that are little-o of other functions at the neighbourhood defined by some filter, and a sigma type `littleo_type` that bundles a function with the proof of this predicate. Similarly for big-$\mathcal{O}$, we provide a predicate `bigO_def` and a type `bigO_type`. The formal definitions of the predicates advantageously use the `near` notation introduced in Section 6.1.1 to encapsulate the existential quantifiers.

```
Context {T : Type} {K : absRingType} {V W : normedModType K}.

Let littleo_def (F : set (set T)) (f : T -> V) (e : T -> W) :=
  forall ε : R, 0 < ε -> \forall x \near F, `|[f x]| <= ε * `|[e x]|.

Let bigO_def (F : set (set T)) (f : T -> V) (e : T -> W) :=
  \forall k \near +oo, \forall x \near F, `|[f x]| <= k * `|[e x]|.
```

Regarding notational conventions in the remainder of this section, note that, like in the code snippet just above, `T` is a type, `K` is a ring equipped with an absolute value, and `V` and `W` are normed modules over `K`; they all are implicit arguments of forthcoming definitions.

The sigma types `littleo_type` and `bigO_type` directly follow from the corresponding ternary predicates:

```
Structure littleo_type (F : set (set T)) (e : T -> W) := Littleo {
  littleo_fun :> T -> V;
  littleoP : littleo_def F littleo_fun e
}.

Structure bigO_type (F : set (set T)) (e : T -> W) := BigO {
  bigO_fun :> T -> V;
  bigOP : bigO_def F bigO_fun e
}.
```

For the sake of readability, we slightly simplified the definitions compared to the source code: we removed `Prop` to `bool` coercions and the `littleoP` and `bigOP` fields are lemmas deduced from unnamed fields in the actual code.

Let us comment more specifically on the structure `littleo_type`. It packs a function, namely the `littleo_fun` projection, with a proof that it is a little-o of `e`, providing us with the type of functions that are a little-o of another function. In particular, we can inhabit this type with the null function (and the trivial proof that it is a little-o). Let us call `littleo0`

this record with the null function. The type `littleo_type F e` is furthermore equipped with the notation `{o_F e}` to improve reading.

So `littleo_type` and `bigO_type` provide a type-theoretic variant of the set-theoretic notation for little-o and big-$\mathcal{O}$, but it can be argued that such a set-theoretic notation is misplaced because it precludes the equational viewpoint that Knuth advocates [Knu98]. It may even be considered anachronistic now that today's students use symbolic algebra systems like Maple [Map19] and SageMath [SD19] where the big-$\mathcal{O}$ notation appears in power series calculations. In this work, we make a strong case for the equational viewpoint. We now explain how to recover it.

Indeed it is also in the folklore to write $f = g + o(e)$ to mean $f - g = o(e)$ in the previous acceptation. Since expressions involving the little-o notation are to be considered as classes of functions, the formula $f = g + o(e)$ suggests a reading in terms of a congruence relation. It might therefore seem like a good idea to formally define the corresponding equality and let it be denoted by a ternary notation. However, doing so carelessly might preclude routine mathematical practice, first because the bound e changes a lot from one equality to another, for example, if

$$f(x) = g(x) + \underset{x\to 0}{o}(x)\,,$$

then

$$xf(x) = xg(x) + \underset{x\to 0}{o}\left(x^2\right).$$

Second, mathematicians add little-o and big-$\mathcal{O}$ from various scales as in: if

$$f(x) = g(x) + \underset{x\to 0}{o}(x)$$

and

$$g(x) = \underset{x\to 0}{\mathcal{O}}\left(x^2\right),$$

then

$$f(x) = \underset{x\to 0}{o}(x)\,.$$

To reflect this mathematical practice, we decided to stress that $f = g + o(e)$ actually means "$f = g + h$ where h is a little-o of e". We thus define

$$f = g + o\,(e) \Leftrightarrow \exists h.\ f = g + o\,(e)\,[h],$$

where $o(e)[h]$ denotes h if h is a little-o of e, and 0 otherwise. As a consequence, the statement $f = g + o(e)[h]$ means $f = g + h$ if h is little-o of e and $f = g$ otherwise. In both cases, $f - g$ is indeed a little-o of e, hence this definition is sound. Note that this definition is also complete: if $f - g$ is a little-o of e, then $f = g + o(e)[f - g]$ holds.

In Coq, to define $o(e)[h]$, we provide a function `mklittleo` that builds a little-o from an arbitrary function. Given a function `h`, `mklittleo` tries to coerce it to the subtype of little-os and, when it fails, it returns the null little-o (using the function `littleo0` mentioned before).

This mechanism of partial projection into a subtype is provided by the generic operator `insubd` from the Mathematical Components library.

```
Definition mklittleo (F : filter_on T) (h : T -> V) (e : T -> W) :=
  littleo_fun (insubd (littleo0 F e) h).

Notation "[o_ x e 'of' h ]" := (mklittleo x h e)
  (at level 0, x, e at level 0, only parsing).
```

Here, the `filter_on` type is a structure meaning that `F` is a filter (and which is meant to replace the `Filter` type class in the near future). For the sake of simplicity, we removed phantom types (used for the inference of the `filter_on` structure) from the definition.

In order to avoid explicitly stating witnesses, we notice that if $f = g + h$, then $h = f - g$ hence h is a little-o of e if and only if $f - g$ is. This leads us to define the sought ternary notation to be:

```
Notation "f = g '+o_' x e" := (f = g + [o_x e of f - g]).
```

The ternary notation `f = g +o_x e` expands to `f = g + [o_x e of f - g]`, but we deliberately hide the `h` in the printing of the notation `[o_x e of h]` so that it prints back `'o_x e`.

However, if we try to prove `f = g +o_x e` in a purely arithmetical way, we might rewrite with equations for `f` and `g` and finally get a goal of the form $o(\mathtt{e}) = o(\mathtt{e})$. In a pen-and-paper proof, this is considered as trivial, but in a formal proof, both little-os hide functions h and h', and the statement to prove is in fact the equality $o(\mathtt{e})[h] = o(\mathtt{e})[h']$. In this situation there is very little chance that this unification succeeds, since there is no reason for h and h' to be equal. Our methodology consists in replacing h' with an existential variable $?h'$ as soon as possible. This is made possible because of the following observation:

$$f = g + o\,(e)\,[f - g] \Leftrightarrow \exists h.\ f = g + o\,(e)\,[h],$$

which allows to replace a goal `f = g +o_x e` with the goal `f = g + [o_x e of ?h]`, where `?h` is an existential variable. This goal is printed `f = g + 'a_o_x e`, once again stressing that `f - g` **is a** little-o of `e`.

6.2.2 Examples and Applications

Our main concern is to preserve the benefits of the equational view of little-o and big-$\mathcal{O}$. This means developing a small theory containing the main "equations" one may need in order to combine them easily. Once sufficiently many equations are proven, this allows the user to prove facts about little-o and big-$\mathcal{O}$ using informal reasoning, without having to go back to the definition of little-o and big-$\mathcal{O}$ and to do explicit local reasoning, except in particular cases where the theory lacks an equation.

We do not claim to have reached such a complete set of equations, but we proved a few equations that seemed important. We first give examples of such equations, illustrating on a proof the benefits of the `near` notations and tactics in this context, and then describe a few applications of our formalisation of Bachmann-Landau notations.

Equational Theory

First, we have arithmetic rules for little-o and big-$\mathcal{O}$. For instance, little-o absorbs addition and the product of a $\mathcal{O}(h_1)$ and a $\mathcal{O}(h_2)$ is a $\mathcal{O}(h_1 \cdot h_2)$.

```
Context {F : filter_on T}.

Lemma addo (f g : T -> V) (e : T -> W) :
  [o_F e of f] + [o_F e of g] =o_F e.

Lemma mulO (h1 h2 f g : T -> R) :
  [O_F h1 of f] * [O_F h2 of g] =O_F (h1 * h2).
```

We also have a few rules combining little-o and big-$\mathcal{O}$. For example, a $o(e)$ is also a $\mathcal{O}(e)$ and a little-o of a $\mathcal{O}(g)$ is a $o(g)$.

```
Lemma littleo_eqO (e : T -> W) (f : {o_F e}) : (f : _ -> _) =O_F e.

Lemma littleo_bigO_eqo (g : T -> W) (f : T -> V) (h : T -> X) :
  f =O_F g -> [o_F f of h] =o_F g.
```

Of course, in order to prove this set of equations, local reasoning is necessary at some point. This is where the `near` tactics from Section 6.1.1 come into use. For instance, let us have a look at the proof of Lemma `littleo_bigO_eqo`.

The function `f` is a $\mathcal{O}$(`g`) and the function `[o_F f of h]` is a o(`f`), either equal to `h` if it is a o(`f`), or to the null function. Since the goal is to prove that the function `[o_F f of h]` is a o(`g`), the first step is to go back to the definition of little-o and introduce the universally quantified ε. This is the application of Lemma `eqoP` at line 1 that recovers the definition `littleo_def` seen in Section 6.2.1.

```
1 move=>; apply/eqoP => _/posnumP[e]; have [k c] := bigO _ g.
```

In line 1, we also replaced ε and the assumption `0 < ε` from `littleo_def` with a canonically positive `e` (thanks to canonical structures we alluded to in the proof in Section 6.1.2) through the `posnumP` view. Moreover, we replaced `f` in `[o_F f of h]` with the function `[O_F g of f]` by rewriting with the assumption of Lemma `littleo_bigO_eqo`. We also abstracted the latter function (thanks to Lemma `bigO` at line 1) using a fresh function `k` and, since it was a big-$\mathcal{O}$ of `g`, we get as a hypothesis a positive constant `c` such that

```
\forall x \near F, `|[k x]| <= c * `|[g x]|.
```

At this point the goal to prove is

```
\forall x \near F, `|[k x]| <= c * `|[g x]| ->
\forall x \near F, `|['o_(x \near F) (k x)]| <= e * `|[g x]|.
```

We use Lemma `filter_app` (recall Section 6.1.1) to combine both statements so that we should now prove

```
\forall x \near F, `|[k x]| <= c * `|[g x]| ->
                   `|['o_(x \near F) (k x)]| <= e * `|[g x]|.
```

Now we give ourselves an `x` which is near `F` thanks to the `near=> x` tactic.

```
2 apply: filter_app; near=> x.
```

We have to prove that

```
‘|[k x]| <= c * ‘|[g x]| -> ‘|['o_(x \near F) (k x)]| <= e * ‘|[g x]|,
```

which we do by manipulating the inequalities until we reach the goal

```
‘|['o_(x \near F) (k x)]| <= e / c * ‘|[k x]|
```

as the result of multiple rewritings and transitivity (see line 3).

```
3 rewrite -!ler_pdivr_mull //; apply: ler_trans; rewrite ler_pdivr_mull // mulrA.
```

The latter goal should be true for `x` which is near `F` since `'o_(x \near F) (k x)` is a little-o of `k` and `e / c` is positive.

To finish the proof, we can now call the `near: x` tactic. At this point, the remaining goal is

```
\forall x \near F, ‘|['o_(x \near F) (k x)]| <= e / c * ‘|[k x]|
```

which can be proven using the filter characterisation of little-o (at line 4).

```
4 by near: x; apply: littleoP.
```

The last line of the proof script calls the `end_near` tactic to dispose of the remaining existential variables:

```
5 Grab Existential Variables. all: end_near. Qed.
```

Application: Asymptotic Equivalence

Two functions f and g are equivalent at a neighbourhood of point a (denoted by $f \sim_a g$) when $f = g + o_a(g)$. Thanks to the ideas explained in Section 6.2.1 and to the already proven equations, the fact that $\sim$ is an equivalence relation can be established by short proof scripts. For the sake of illustration, let us explain how we show that $\sim$ is symmetric and transitive.

The symmetry of $\sim$ is mechanised as follows (`f ~_F g` is the Coq notation for $f \sim_F g$):

```
Context {F : filter_on T}.

Lemma equiv_sym (f g : T -> V) : f ~_F g -> g ~_F f.
Proof.
move=> fg; have /(canLR (addrK _))<- := fg.
by apply: eqaddoE; rewrite oppo (equivoRL _ fg).
Qed.
```

The first line of the proof is made of standard tactics that change the goal to $f - o(g) \sim f$. Lemma `eqaddoE` implements the idea described in Section 6.2.1: it introduces an existential variable $?h$ such that the goal becomes $f - o(g) = f + o(f)[?h]$. Rewriting with Lemma `oppo` turns $f - o(g)$ into $f + o(g)$ and Lemma `equivoRL` turns $o(g)$ into $o(f)$ (it uses the hypothesis $f \sim g$). The right- and left-hand sides can now be unified and the proof is completed.

The transitivity of $\sim$ is mechanised as follows.

```
Lemma equiv_trans (f g h : T -> V) : f ~_F g -> g ~_F h -> f ~_F h.
Proof.
by move=> -> ->; apply: eqaddoE; rewrite eqoaddo -addrA addo.
Qed.
```

After the application of Lemma `eqaddoE`, the goal is

$$h + o\,(h) + o\left(h + o\,(h)\right) \sim h + o\,(h)\,[?e],$$

where $?e$ is an existential variable.

Lemma `eqoaddo` transforms $o(h+o(h))$ into $o(h)$ and Lemma `addo` transforms $o(h)+o(h)$ into $o(h)$. After rewriting, the goal is $h + o(h) \sim h + o(h)[?e]$, so that unification succeeds and completes the proof.

Application: Differential of a Function

Thanks to Bachmann-Landau notations, we can define differentials using the `get` function from Section 5.3.3. As we discussed in Section 4.3.2, in Coquelicot there is only a function computing the derivative of a real function at any point. Thanks to the `get` function, it is possible to define a function computing the differential of any function from a normed module to another one at any neighbourhood defined by a filter.

The differential $\mathrm{d}\,f_x$ of a function f at point x is the unique continuous linear operator that satisfies the following asymptotic development

$$f(x+h) = f(x) + \mathrm{d}\,f_x(h) + \underset{h\to 0}{o}\,(h)\,,$$

or equivalently

$$f(h) = f(x) + \mathrm{d}\,f_x(h-x) + \underset{h\to x}{o}\,(h-x)\,.$$

Thanks to our formalisation of Bachmann-Landau notations, we can directly use this definition through the `get` function.

```
Definition diff {K : absRingType} {V W : normedModType K} (F : filter_on V)
  (f : V -> W) :=
  (get (fun (df : {linear V -> W}) => forall x,
  f x = f (lim F) + df (x - lim F) +o_(x \near F) (x - lim F))),
```

We provide the notation `'d f F` for the differential of `f` at the neighbourhood defined by `F`.

We ported the inference mechanism from Section 3.3.3 to this new definition (and to the definition of derivatives from Section 5.3.3). We also wrote an equational theory for differentials and derivatives.

A notable point in this theory is the chain rule for differentials,

$$\mathrm{d}(g\circ f)_x = \mathrm{d}\,g_{f(x)} \circ \mathrm{d}\,f_x,$$

whose statement in Coq looks like its pen-and-paper equivalent thanks to our notations.

```
Lemma diff_comp (U V W : normedModType R) (f : U -> V) (g : V -> W) x :
  differentiable f x -> differentiable g (f x) ->
  'd (g \o f) x = 'd g (f x) \o 'd f x.
```

The equational theory of Bachmann-Landau notations we developed made it possible to write a very compact proof for this theorem.

Application: Uniform Big-$\mathcal{O}$

Boldo et al. designed a notion of uniform big-$\mathcal{O}$ in their work on the numerical resolution of the wave equation [BCF+13]. They are interested in relations of the form

$$f(x, \Delta) = \underset{\Delta \to 0}{\mathcal{O}}(g(\Delta)),$$

but with a uniform definition with respect to x, i.e. with the following definition:

$$\exists C > 0.\ \exists \alpha > 0.\ \forall x.\ \forall \Delta.\ \|\Delta\| < \alpha \Rightarrow \|f(x, \Delta)\| < C \|g(\Delta)\|.$$

We can formalise this uniform big-$\mathcal{O}$ as

```
(fun p => f p) =O_F (fun p => g p.2)
```

where `p` represents the pair (x, Δ) and `p.2` is thus Δ, with an appropriate choice for the filter `F`. The filter `F` must be a filter on a Cartesian product and can be defined as a product filter (recall the end of Section 6.1.1). The first filter in this product corresponds to x. Since there is no constraint on x, we can choose the trivial filter containing only the total set. The second filter in this product, which corresponds to Δ, is the neighbourhood filter of 0 (i.e. `locally 0`).

In fact, in their source code, Boldo et al. force Δ to be within a particular domain and use sigma types instead of the existential quantifier. We can still provide an equivalent definition thanks to a "restriction" operator on filters that already existed in Coquelicot (the `within` function), and prove that it is indeed equivalent to their definition.

Part II
Designing a Library of Mathematics

CHAPTER 7

EVALUATION OF OUR LIBRARY

In this chapter, we give an early assessment of the Mathematical Components Analysis library [ACM+]. This assessment is mainly based on our case study on the inverted pendulum: we reproduced the proofs from Part I using our library instead of Coquelicot (these new proofs are released as a new version of the repository of our case study [CRb]).

We first describe the improvements on our original proofs in Section 7.1. Then, we discuss issues of our library in Section 7.2. Finally, we present related work in Section 7.3.

7.1 Improvements on our Case Study

We reproduced our case study using Mathematical Components Analysis in order to gather data for an evaluation of the library. Not counting the elements that have been integrated into the library, both formalisations roughly have the same size: the formalisation based on Mathematical Components Analysis is around a hundred lines of code shorter. We may however pinpoint some reasons why there is a slight improvement.

First, the library was designed for classical reasoning, bearing in mind the issues described in Section 4.3.1. This means that our proofs are closer to the pen-and-paper ones because we do not need to adapt the definitions to the logical context.

Moreover, our axioms allow for the use of the algebraic theories of Mathematical Components [MCT] on the type `R` of real numbers. These theories are well-thought and designed for efficient theorem proving, which compensates for the limitations of Coq's standard library. In particular, the compatibility with Mathematical Components improves the way we deal with comparisons of expressions. The comparison predicates in Mathematical Components compute boolean values, which makes it possible to use small-scale reflection, and come with a battery of lemmas that make proof writing efficient. Furthermore, most positivity proofs are automated thanks to a set of canonical structures that Cyril Cohen developed

in the library. As a consequence, all the proofs that involve comparisons on real numbers are shortened.

The new tools in the library are also sources of improvements. The theory of supremum we mentioned in Section 5.4.3 contains the exact theorems that we needed for our proofs. Moreover, the `near` tactics play a significant role: a great part of the formalisation is based on the manipulation of filters, which is made easier by these tactics (recall our discussion in Section 6.1.2). Finally, we make use of the `diff` function from Section 6.2.2, but it has a relatively limited impact since we mostly manipulate derivatives of functions defined on `R` and not their differential.

7.2 Remaining and New Issues

Although our new library resolves some issues we discussed in Section 4.3, some of them still remain, and new ones appeared.

First, we still lack a good theory of iterated maximum on families of real numbers. It is not obvious to solve this issue because, as it stems from our discussion in Section 3.3.1, it boils down to a problem of subtyping. To our knowledge, there is no good mechanism to deal with subtypes in Coq. The type-correctness conditions (TCC) from PVS [SO99] may be a good source of inspiration.

Then, although the `near` tactics ease filter manipulation, it is still necessary to split the epsilons in proofs. Fortunately, this is not a big constraint since the user does not have to know beforehand how the epsilons will be split. However, we received comments from users who found the tactics hard to use. Thus, we still have to work both on the documentation and on the framework to make it more intuitive.

About new issues, we already mentioned in Section 5.4.2 the lack of practicality of the equivalent of ball splitting for entourages. More precisely, the proofs on uniform spaces that involve ball splitting are a bit longer with entourages in the current state of the experiment. Proofs on normed modules are not impacted by this change of definition since we may use balls defined by the norm instead of entourages. We do not know yet how problematic this issue is.

Finally, the biggest issue of our new library is also an issue of the Mathematical Components library. Indeed, although a strength of the Mathematical Components library is the possibility to perform small computation steps to do proofs thanks to small-scale reflection, it is impossible to do so for larger computations on its algebraic structures. Indeed, data structures in Mathematical Components contain locks that prevent computation and, even without locks, these data structures are not appropriate for efficient computation. These locks are essential because they make notations steady (e.g. for big operators) and they make unification more efficient, which is important since unification is the base of the inference system in the hierarchy of Mathematical Components (recall Section 5.1.2). The data structures cannot be replaced with computationally more efficient ones because they are specifically designed to do proofs in an efficient way.

Proof by computation can be emulated through the use of rewriting rules given by the axioms defining the behaviour of arithmetic operations, or it can be done using more complex techniques such as large-scale reflection (which we discuss in Section 9.1). In our formalisation from Part I, we were able to use the `ring` [GM05] and `field` [DM01] tactics to this end. The

layers of structures and notations added by Mathematical Components however make it hard to use them: these tactics fail to recognise the ring (respectively field) of real numbers because they cannot interpret the definitions from Mathematical Components.

Several options are available but none is perfect. First, it is possible to declare new ring (respectively field) structures to help the tactics interpret the definitions correctly. But the multiplicity of structures in Mathematical Components makes it hard to cover every possible case: definitions from several structures may appear in a unique expression, there is an overloading of notations, and the tactics only allow for one interpretation for each operation.

Then, Strub developed two tactics in Jasmin [ABB+17], `ssring` and `ssfield`, that replace the `ring` and `field` tactics for Mathematical Components, based on a different implementation of reification (we explain reification in Section 9.1). However, the `ssfield` tactic fails in our proofs: it does not terminate in a reasonable time. Hence, instead of calling `ssfield`, we have first to remove each fraction from our goal by multiplying the goal by the common denominator of all fractions and then call `ssring` instead, which makes proof scripts longer.

Finally, we also worked on a prototype of tactic that should generalise the `ring` tactic (see Chapter 9), and that could be extended to generalise the `field` tactic too, but it is still in an early stage of development and it is not usable yet.

7.3 Related Work

We first focus on existing libraries for analysis in Section 7.3.1. Then we discuss related work on Bachmann-Landau notations in Section 7.3.2. Finally, we present in Section 7.3.3 tactics that were designed with the same goal as our `near` tactics: delaying the production of explicit witnesses for existential propositions.

7.3.1 Libraries for Analysis

The main reference to discuss analysis libraries is the survey by Boldo et al. [BLM16]. Instead of repeating its content here, which is also detailed in Lelay's PhD thesis [Lel15], we invite the interested reader to refer to these two presentations. We will just complete the picture with libraries that are not presented in these works and try to make clear the position of Mathematical Components Analysis with respect to these libraries.

We already discussed libraries in Coq in Section 2.3.1, but we will add a few comments on Coquelicot. As shown by Section 5.2, Coquelicot is truly the base of our library. Most definitions in our library come from Coquelicot and those that differ from Coquelicot's ones were developed for two main reasons: we work in a different logical settings and we adapted definitions to new tools (e.g. the `near` tactics) that were not in Coquelicot. On the other hand, several parts of the Coquelicot library have not been adapted to our new context yet (mostly, sequences, integrals and series).

The Coquelicot library also provides total functions to compute the limit and the derivative of a function. They are however restricted to functions from $\mathbb{R}$ to $\mathbb{R}$. We define a limit function for any function whose domain and codomain are equipped with canonical filters and a differential function for any function whose domain and codomain are normed modules. The crucial difference is that we include the existence of choice functions in our hierarchy at the

cost of additional axioms, which give us these functions for free, while in the Coquelicot library they are constructed from the limited principle of omniscience.

This difference in terms of logic is very relevant here and we shall precise this point in light of Table 1 from the survey by Boldo et al. [BLM16]. Coquelicot is **constructively** built on top of a classical axiomatisation of the set of real numbers [May01], which implies that the parts of the survey concerning the logic of Coq's standard library are still true for Coquelicot. In comparison, thanks to our axioms from Section 5.1.1, Mathematical Components Analysis falls in with the systems based on higher-order logic: HOL4 [SN08], ProofPower-HOL [Lem06], HOL Light [Har16] and Isabelle/HOL [NPW02]. Indeed, we work in a classical settings where Hilbert's epsilon operator [Hil22] is available. Moreover, although this is not shown by Table 1 in this survey, these systems have an equality which is extensional on functions, which we obtain thanks to the functional extensionality axiom.

This is not the only point of similarity with the systems based on higher-order logic: we also provide in our library tools to deal with analysis in higher dimensions. Although the `NormedModule` interface theoretically covers higher-dimensional vector spaces, and even spaces of infinite dimension, Coquelicot is limited on concrete types to dimension 2. Thanks to matrices, we may deal with any finite dimension. The difference with the systems based on higher-order logic is that we benefit from dependent types through the implementation of vectors in Mathematical Components.

Finally, there is also a part of the Lean Mathematical Components library [LMCLD] which is dedicated to analysis. It contains an extensive hierarchy of structures, that covers most of ours. It is also developed on classical axioms and exploits filters for the formalisation of convergence. In this library, uniform spaces are defined using entourages (recall Definition 5.5).

7.3.2 Related Work on Asymptotic Reasoning

The Coquelicot library contains ternary predicates defining little-o and asymptotic equivalence of functions. Our definitions are basically the same (in particular the ternary predicate `littleo_def`) but their theory is not quite developed in Coquelicot. We provide a set of notations and a more substantial equational theory on top of our definitions, which makes them easier to manipulate. We also have notations and a theory for big-$\mathcal{O}$.

Avigad and Donnelly's formalisation in Isabelle/HOL [AD04] views big-$\mathcal{O}$ as sets. They describe inclusion and equational reasoning on big-$\mathcal{O}$ at the set level, and they manage to prove the prime number theorem using it. Thirteen years later, Eberl improves and extends their work by providing, in addition to big-$\mathcal{O}$, the little-o, Ω, ω, and Θ notations, in order to prove the complexity of divide-and-conquer algorithms [Ebe17]. Coupled with Isabelle/HOL's heavy automation, his Landau symbols halve the size of his proofs. He uses in particular this formalisation to develop tactics that automatically compute (and prove) asymptotic developments of sequences given by recurrence relations [Ebe19b] and of combinations of standard real functions [Ebe19a].

Guéneau et al. [GCP18] have developed in Coq a library to formalise the time complexity of OCaml programs, later applied to a cycle detection algorithm [GJCP19]. To represent asymptotic bounds, they provide a formalisation of the big-$\mathcal{O}$ notation. Similarly to us, their definition relies on filters, but only on finite filter products of the `eventually` filter (the equivalent of `Rbar_locally p_infty` on `nat`, recall Section 2.3.2) and its equivalent in $\mathbb{Z}$. Furthermore, they define a type for types equipped with **one filter**, while we make it possible

to have **a different filter for each element of the type** thanks to the `filteredType` structure.

However, in the face of the difficulties encountered to reproduce the (apparently sloppy) manipulation of the big-$\mathcal{O}$ notation, they give up on producing proofs "as simple [...] as their paper counterparts", choose to formalise the big-$\mathcal{O}$ notation as a dominance relation, and deprive themselves of Coq's equational reasoning capabilities. Their library would require extension with the little-o notation and to arbitrary filters for it to "have other applications in mathematics". In comparison, our work already provides both notations, retains equational reasoning, and already fits together with a hierarchy of mathematical structures designed on the model of Mathematical Components [GGMR09, MT18].

A particular formalisation of big-$\mathcal{O}$ to be mentioned is the one by Boldo et al. [BCF+13]. Their uniform big-$\mathcal{O}$ can be expressed with our definition through the choice of the appropriate filter, as described in Section 6.2.2.

Finally, Avigad also developed in Lean [LMCLD] a library for little-o and big-$\mathcal{O}$. They are defined as ternary predicates equivalent to our `littleo_def` and `bigO_def` predicates once the `near` notations are unfolded.

7.3.3 Related Work on Delayed Production of Witnesses

Eberl's Landau symbols [Ebe17] are defined using the `eventually` construct of the standard library of Isabelle/HOL [HIH13] that applies a predicate to a filter. Formal proofs therefore enjoy the `eventually_elim` tactic that automates the application of filter-related lemmas together, and is often combined with other lemma collections (such as `field_simps`). The `eventually_elim` tactic is a simpler form of the `near` tactics, well adapted to Isabelle/HOL's proof style. Indeed, when using `eventually_elim`, one lists upfront a list of hypotheses that will be used by the automated proof search. Using `near`, these sets are **inferred** at the appropriate places while writing the proofs in an imperative style.

Guéneau et al.'s proofs on complexity [GCP18] also use delayed production of witnesses of existential quantifiers in the particular case of the computation of cost functions. They use Procrastination [Gué18], a small library of tactics similar to our `near` tactics. The main difference between Procrastination and the `near` tactics is the following. To prove a given goal using Procrastination, one can introduce variables and accumulate properties about them. Once the goal is proven, the user is asked to provide an actual value for these variables which satisfies every accumulated property. Our work is more centred on filters: we do not have to provide a witness of the satisfiability of a predicate, but only to prove that the accumulated predicates belong to a given filter. The implementation of Procrastination also has more tactics, which are more complex, while we try to minimise them, following the small-scale reflection strategy [MT18].

Both our work and the Procrastination library are a generalisation of the `bigenough` library by Cohen [Coh12]. This library only deals with statements that are eventually true in $\mathbb{N}$: it is a special case of the `near` tactics for the `eventually` filter.

Part III

Tools for Automation

Part III
Tools for Automation

CHAPTER 8

REFINEMENT AND COMPUTATION

As we explained in Section 7.2, in the Mathematical Components library [MCT] and hence in Mathematical Components Analysis [ACM+], some data structures make computation impossible. This is a real issue when one tries to formalise proofs where computation plays a significant role. In this chapter, we describe a framework that makes it possible to do certified and efficient computation by changing both the representation of the objects that are manipulated and the algorithms that are executed.

This framework is based on a technique called *refinement* (see Section 8.1). We describe in Section 8.2 the benefits of this technique in the context of proofs and a tactic we designed in order to ease the use of refinement in proofs. We also explain how to automate parts of the refinement process thanks to *parametricity* (see Section 8.3).

Our description of refinement is based on the work of Dénès et al. [DMS12], later improved by Cohen et al. [CDM13], who introduced the use of parametricity for refinement. Our only original contributions in this chapter are the tactic we describe in Section 8.2.3, which was designed in collaboration with Cyril Cohen and which generalises the one we published [CR17b], and our current work on parametricity in collaboration with Cyril Cohen, Assia Mahboubi, Matthieu Sozeau and Nicolas Tabareau, briefly explained in Section 8.3.3. All code snippets come from the CoqEAL library [CCD+], unless otherwise specified.

8.1 Refinement

Refinement [Wir71, Hoa72] is a term usually used to describe a step-by-step approach in the verification of a program. We discuss here another kind of refinement [DMS12], which is nevertheless in some ways analogous to the standard notion (see Section 8.1.1). We distinguish in particular two kinds of refinement: *program refinement* (see Section 8.1.2) and *data refinement* (see Section 8.1.3), which can be composed (see Section 8.1.4).

8.1.1 Definition of Refinement

Refinement, as defined by Wirth [Wir71] and later integrated by Hoare in a methodology for program verification [Hoa72], makes it possible to obtain a certified implementation of a given function through simple steps. One starts with an abstract representation of the function and its specification. Then, the function is progressively refined to a concrete program that implements it by changing the representation of the data structures and the algorithm, going from high-level representations to low-level ones. Each step is proven correct with respect to the previous one.



Example

A specification may be given in the form of a Hoare triplet [Hoa69]:

$$\vdash \{\mathit{precondition}\}\, P\, \{\mathit{postcondition}\},$$

where P is abstract and represents the program to be implemented. For instance, one may start with the following specification:

$$\vdash \{x = M\}\, P\, \{y = \det M\}.$$

Here, the program takes as input a matrix M, stored in variable x, and has to compute the determinant of M and to store its value in variable y. We do not specify here how matrices are represented, nor which is the ring of coefficients.

Then, one designs and proves correct with respect to this specification an algorithm computing determinants (e.g. Bareiss' algorithm [Bar68]). Note that the representations of matrices and coefficients are still abstract: the only required primitives, which are assumed correct at this point, are accesses to coefficients at given coordinates in a matrix and arithmetic operations in the ring of coefficients.

The next step is to fix the representation of matrices. This is when the programming language has to be chosen. In a programming language featuring arrays, one may use them to represent matrices. One has to implement the primitives that are needed by the algorithm based on this representation and to prove that they are correct. For the sake of modularity, the ring of coefficients stays abstract.

The last step is to choose the representation of coefficients. Assume one is only interested in determinants of matrices on integers. One may then choose to use an implementation of the binary representation of integers in the chosen programming language. Once again the implementation of the primitives has to be proven correct.

All in all, we obtain a certified program that computes the determinant of a given matrix on integers.

Similarly to the standard notion of refinement, we are interested in a technique that allows for a change of representation of programs. We also want this change to be certified: two representations of a program must compute the same function, i.e. they must be extensionally equal.

However, although some sort of compositionality applies (see Section 8.1.4), this is not a step-by-step approach: the change of representation is performed in one go. Moreover, as opposed to the mechanism implemented in the Isabelle/HOL code generator [Lam13, LL19]

or in Coq's Fiat library [DPGC15], the framework which is of interest to us, implemented in the CoqEAL library [CCD+], does not relate an abstract representation to a concrete implementation but two concrete implementations.

The specificity of the refinement framework in CoqEAL is that it relates proof-oriented implementations to computation-oriented ones. The point is to benefit from proof-oriented representations, such as the ones in the Mathematical Components library [MCT], to prove correctness properties while still being able to perform computations on more efficient representations. By proving correct the computation-oriented implementations, only with respect to the proof-oriented ones, we know that the result of an efficient computation is the computation-oriented representation of the proof-oriented value on which we proved correctness.

The correctness of a representation with respect to another one is expressed through a *refinement relation*. For instance, if one wants to relate the unary representation of non-negative integers (`nat` in Coq) to the binary one (`N` in Coq), a possible refinement relation is the following one:

```
Definition Rnat (n : nat) (m : N) := nat_of_bin m = n,
```

where `nat_of_bin` is a (trusted) function computing the unary representation of a binary non-negative integer. The intended meaning of `Rnat n m` is "`n` and `m` represent the same (abstract) non-negative integer".

8.1.2 Program Refinement

Program refinement consists in replacing an algorithm with a different one that computes more efficiently the same function, but **preserving the data structures**. The correctness of such a refinement thus amounts to proving that the two algorithms are extensionally equal: an algorithm `Q` is a refinement of another algorithm `P` if and only if

```
forall x, P x = Q x.
```

Example

In Mathematical Components, the determinant of a matrix on a ring is defined through the `\det` function, which implements the Leibniz formula

$$\det M = \sum_{\sigma \in \mathfrak{S}_n} \varepsilon(\sigma) \prod_{i=1}^{n} M_{\sigma(i),i},$$

where $\varepsilon(\sigma)$ is the signature of the permutation σ and n is the dimension of the matrix M.

Locks on the matrix structure put aside, this gives a highly inefficient program computing the determinant of a matrix. Fortunately, CoqEAL contains an implementation of Bareiss' algorithm, named `bdet`.

Its correctness property is the following equality.

```
Lemma bdetE (R : comRingType) (n : nat) (M : 'M[R]_(1 + n)) :
  bdet M = \det M,
```

where `comRingType` is MATHEMATICAL COMPONENTS' structure of commutative rings and `'M[R]_(1 + n)` denotes the type of square matrices on `R` of size `1 + n`.

8.1.3 Data Refinement

Data refinement involves putting in correspondence the base objects of two different data representations, but also the primitives of these data structures (thus, through some kind of program refinement). We already presented a relation for such a refinement (`Rnat` in Section 8.1.1). We give here an example based on matrices.

Example

As explained in Section 3.3.1, matrices are represented in MATHEMATICAL COMPONENTS by a structure which essentially packs a sequence with a proof on its length. Some matrix operations however affect the size of the matrix (e.g. matrix product transforms matrices of sizes $m \times p$ and $p \times n$ into a matrix of size $m \times n$).

A more efficient representation of matrices is thus the sequence alone, without the proof and with no information about the size of the matrix. This representation is parametrised by the type of coefficients.

```
Definition seqmx {A : Type} := seq (seq A).
```

For example, the zero matrix of size $m \times n$ is then the sequence that contains m times the sequence that contains n times the null element of the ring of coefficients.

```
Definition seqmx0 {A : Type} `{zero_of A} (m n : nat) :=
  nseq m (nseq n 0%C).
```

Here, the `0%C` notation denotes the null element of `A`, inferred thanks to the `zero_of` type class (see Section 8.2.2 for a description of the inference mechanisms for refinement).

This structure is then proven correct with respect to the matrix structure from MATHEMATICAL COMPONENTS when it is instantiated on the proof-oriented `ringType` structure. The refinement relation relates a matrix `M` to a sequence `s` by stating that `s` and the sequences that it contains have the right sizes (corresponding to the dimensions of `M`) and that corresponding coefficients in `s` and `M` are equal. We give the correctness property for the zero matrix.

```
CoInductive Rseqmx {R : ringType} {m n : nat} :
  'M[R]_(m,n) -> @seqmx R -> Type :=
  Rseqmx_spec M s of
    size s = m
  & forall i, i < m -> size (nth [::] s i) = n
  & forall i j, M i j = nth 0%C (nth [::] s i) j : Rseqmx M s.

Lemma Rseqmx_0 (R : ringType) (m n : nat) :
  Rseqmx (const_mx 0%R) (@seqmx0 R _ m n).
```

8.1.4 Composition of Refinements

The essential property for refinement is compositionality: it should be possible to combine several data refinements, or program refinement with data refinement, in a seamless way. For instance, we should be able to compose refinements from dense to sparse polynomials and from unary to binary integers to obtain a refinement from dense polynomials over unary integers to sparse polynomials over binary integers.

In order to achieve this goal, Cohen et al. [CDM13] propose the following refinement methodology:

1. parametrise the algorithm by the data it manipulates using abstract types and abstract basic operations (this is called generic programming).

2. prove the correctness of the algorithm instantiated on a proof-oriented representation of the parameters.

3. use the parametricity of the algorithm to deduce its correctness when it is instantiated on a corresponding computation-oriented representation of the parameters.

This methodology also applies to data structures and their primitives.

In particular, for the composition of program refinement and data refinement, simple extensional equality is no longer sufficient. Indeed, one cannot write

```
forall x, P x = Q x
```

when `P` and `Q` operate on different data structures. In particular, `x` can be the argument of only one of those programs: the one which manipulates the data structure which serves to implement `x`.

A more complex notion of correctness, based on refinement relations, is then used. Keeping in mind the fact that for a refinement relation `R`, `R x y` means "`x` and `y` implement the same object", one may express the correctness of `Q` with respect to `P` in the following way: for all `x` and `y` such that `R x y`, `P x` and `Q y` are related by the appropriate refinement relation (corresponding to the output data types and denoted by `R'` below). This folds into the following statement in CoqEAL:

```
(R ==> R') P Q.
```



Example

Lemma `bdetE` in Section 8.1.2 corresponds to the second step of the methodology for the refinement of the determinant function (the first step being the definition of `bdet` through generic programming).

The last step requires first to have a compositional data refinement for matrices. The two first steps of the refinement of matrices are described in Section 8.1.3. The last one makes it possible to compose this refinement with a refinement of the coefficients. A new refinement relation is defined.

```
Definition RmxC {R : ringType} {C : Type} (rC : R -> C -> Type)
  {m n : nat} : 'M[R]_(m,n) -> @seqmx C -> Type :=
  (Rseqmx \o (list_R (list_R rC)))%rel.
```

Here, the notation `(_ \o _)%rel` represents relation composition (the composition of R and S is $R \circ S = \{(x, y) \mid \exists z.\ (x, z) \in R \wedge (z, y) \in S\}$), `rC` denotes a refinement relation between `R` and `C`, and `list_R` defines a relation on sequences by mapping the relation in argument on their elements.

The true refinement theorem on the zero matrix is then the following one:

```
Lemma RmxC_0 (R : ringType) (C : Type) (rC : R -> C -> Type) ...
  (m n : nat) : RmxC rC (const_mx 0%R) (@seqmx0 C _ m n),
```

where we omitted hypotheses about the null element of `C`.

We can now state the refinement theorem for the determinant:

```
Lemma refine_det (R : comRingType) (C : Type) (rC : R -> C -> Type) ...
  (n : nat) : (RmxC rC ==> rC) \det bdet.
```

We omitted parts of the statement that are related to automation and that we will discuss in Section 8.2.2. In particular, algebraic operations on `C` and their correctness properties are omitted.

8.2 Using Refinement in Proofs

The refinement mechanism we presented in Section 8.1 makes it possible to perform certified and efficient computations inside the proof assistant. We will now explain how to exploit this possibility in order to simplify the proof process.

We first discuss in Section 8.2.1 the interactions between proofs and computation in the Coq proof assistant [CDT19]. Then, we describe the inference mechanism that makes it possible to automate refinement, thus making the use of this mechanism easier (see Section 8.2.2). Finally, we present in Section 8.2.3 a tactic we designed in order to use refinement to simplify expressions inside a goal.

8.2.1 On Proofs and Computation

Computation may be used in the context of proofs in order to shorten and simplify them. We described in Section 3.2 a proof where computation plays a major role: arithmetic expressions need to be simplified in order to derive properties such as sign conditions. This is however not the only use of computation for proofs. Indeed, specialised algorithms, decision procedures, may be used to decide whether an object has a given property. Executing such algorithms, hence performing a computation, is a way of proving theorems.

In this section, we briefly discuss a terminology coined by Barendregt and Cohen [BC01] to describe the different possibilities to make use of computation in a proof assistant. We focus in particular on the options that are available in Coq.

Three strategies are possible to carry out computation in a proof assistant: the *believing*, *sceptical* and *autarkic* approaches.

The Believing Approach

The believing approach is the most efficient one. Computation is performed by an external tool which was designed for it. For instance, one could use a computer algebra system such as Maple [Map19] or SageMath [SD19].

An interface between the proof assistant and the external tool has to be designed. When one wants to compute an arithmetic expression for instance, the first step is to translate the expression (written in the language of the proof assistant) to the language of the computer algebra system. Then, the computation is performed and the result is translated back into an expression that is legible by the proof assistant (see Figure 8.1).

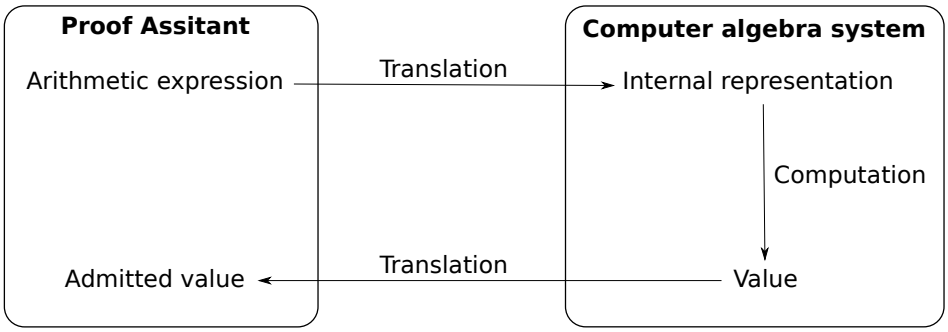


Figure 8.1 – The Believing Approach

The believing approach is named after the fact that any result returned by the external tool is assumed correct. The translation functions are trusted too. This means that this approach gives no guarantee on the correctness of any proof that relies on such computations.

It is possible to have more confidence in this approach if the external tool was previously proven correct, even though the translation functions still have to be trusted. Indeed, one could for instance certify a program in Coq and then extract a program that can be compiled and executed outside the proof assistant (see Letouzey's work [Let04, Let08] for details about Coq's extraction mechanism). Still, this adds to the picture another trusted element: the extraction mechanism is only partially certified as of today [Glo09, Glo12, MPW+18].

The Sceptical Approach

The sceptical approach adds reliability at the cost of efficiency. With this approach, we try to minimise the cost in efficiency: computation is still performed by an external tool, but it is no longer trusted. The proof assistant verifies the results provided by the computer algebra system.

The performance of this approach will be determined by how this verification is performed. The best case scenario is when the external tool is able to provide a *certificate*: together with the result of the computation, the computer algebra system sends to the proof assistant a description of elementary steps that led to this result (see Figure 8.2). The proof assistant can then check the correctness of each of these steps.

In the worst case scenario, only the result of the computation is provided. The proof assistant then has to check the result by itself (or with the help of a human operator), using the autarkic approach. Due to the current limitations of proof assistants (bad performances in

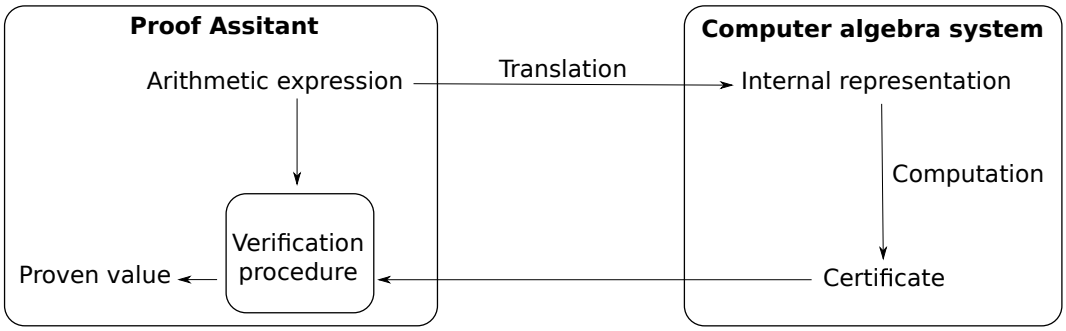


Figure 8.2 – The Sceptical Approach with Certificates

computations, limited number of decision procedures already implemented), only some kinds of computation can be verified without certificates. In Coq for instance, Maple was used as a tool in a formal proof of Apéry's Theorem [CMST14] and an automated interface with Maple was developed for computations in fields [DM05] and quantifier elimination over algebraically closed fields [DM06].

Remark

In this approach, the translation function and the certificate generator do not need to be trusted: if they are incorrect and provide a false result, the verification step performed by the proof assistant will fail and the result will not be used.

Thus, we benefit from the reliability of the proof assistant while keeping part of the efficiency of the computer algebra system.

The Autarkic Approach

With the autarkic approach, we try to fulfil the dream that proof assistants become unified frameworks for mathematics, in which it is possible not only to mechanise proofs but also to carry out certified and efficient computation. Thus, computation is performed inside the proof assistant. This is currently the most inefficient strategy.

The basic mechanism for computation in Coq is conversion (recall our discussion in the introduction). Indeed, Coq's language is based on λ-calculus, which has a computational model. Thanks to the reduction rules of λ-calculus, λ-terms are programs that one may want to execute. Through conversion, Coq captures equality by computation. The `compute` tactic implements this computational model.

In order to make computation more efficient than using Coq's interpreter, other reduction strategies were developed. The `vm_compute` tactic [GL02] improves on the `compute` tactic using a virtual machine that carries out reduction in an optimised way. The `native_compute` tactic [BDG11] further improves efficiency by compiling the goal to OCaml [LDF+18] and using the OCaml compiler to produce binary code that is executed to carry out the computation.

However, conversion may be blocked through locks, as it is the case in the Mathematical Components library [MCT]. A possible work around is to use rewriting rules: with an appropriate set of lemmas, one can capture the computational behaviour of functions with

equalities that can then be used through the `rewrite` tactic. This can however be extremely tedious and fairly inefficient.

The solution proposed in the CoqEAL library [CCD+] is to still make use of conversion, not by tuning the system to remove locks since the underlying structures and algorithms are inefficient, but by using its refinement mechanism in order to carry out computation on more efficient structures, with more efficient algorithms.

8.2.2 Automation of Refinement

The ease of use of the refinement framework is an important concern to us: we want to be able to trigger computation in a seamless way during a proof. Usually, there is no refinement theorem that gives the refinement of the particular expression one wants to refine. It is often necessary to combine several theorems.

For instance, if one wants to refine the expression $2 + 3$, one has to look for refinements of 2 and 3 and for a refinement of the addition on non-negative integers. In short, one has to decompose the expression to be refined into smaller bricks for which refinement theorems are available. Moreover, there is also an overloading issue: the + symbol denotes any addition in a ring. Since different data types have different refinements, it is necessary to use types to know which theorem to apply.

In order to simplify the use of refinement, the CoqEAL library provides an inference mechanism [CDM13] based on type classes [SO08]. Type classes serve two purposes here: they make it possible to resolve overloading of notations and to keep a data base of refinement theorems.

Type Classes for Overloading

In order to achieve overloading of notations, standard operations are bundled into type classes that come with standard notations. For instance, the addition operation is inferred thanks to the `add_of` type class, that comes with the standard `+` notation. It can for example be instantiated for non-negative binary integers.

```
Class add_of (A : Type) := add_op : A -> A -> A.

Notation "+%C" := add_op.
Notation "x + y" := (add_op x y) : computable_scope.

Instance add_N : add_of N := N.add.
```

The interpretation scope `computable_scope`, delimited by `%C`, distinguishes computation-oriented operations from proof-oriented ones, usually given in the scope `ring_scope` from Mathematical Components. Thus, the refinement theorem for addition on non-negative integers is easily written as follows.

```
Lemma Rnat_add : (Rnat ==> Rnat ==> Rnat) addn +%C.
```

Such type classes play a significant role in our running example of Section 8.1. The omitted arguments of Lemma `refine_det` in Section 8.1.4 include arithmetic operations on the computation-oriented type `C`, which are inferred thanks to these classes. Moreover, the `zero_of` argument of `seqmx0` in Section 8.1.3 is such a class.

Refinement Inference

By using a tag to bundle refinement relations into a type class, one can feed type class inference with all the theorems involving these relations. Together with particular instances defining rules to guide the inference, this gives a logic program computing refinements.

```
Class refines {A B : Type} (R : A -> B -> Type) (a : A) (b : B) :=
  refines_rel : R a b.
```

The `refines` class takes as argument the proof-oriented type `A`, the computation-oriented type `B`, a relation between those two types and bundles a proof that two elements `a` and `b` of these types are in relation. By default, inference is guided by the form of `a`: the intuition is that we refine a term of type `A` into a term of type `B` in order to perform a computation. From a logic programming point of view, `a` is the input of the program whereas the other arguments are the outputs.

The refinement theorem for the addition on non-negative integers is for example actually stored thanks to the following instance.

```
Instance Rnat_add : refines (Rnat ==> Rnat ==> Rnat) addn +%C.
```

Once the refinement theorems for the base operators/objects of a data type are stored, Coq can combine them using rules defined by particular instances in order to infer refinement theorems for more complex expressions defined from these smaller bricks.

Among these rules, there is the one for the application of a function. In short, a function applied to an argument is refined to a refinement of the function, applied to a refinement of the argument.

```
Instance refines_apply (A B : Type) (R : A -> B -> Type)
  (C D : Type) (R' : C -> D -> Type) :
  forall (c : A -> C) (d : B -> D), refines (R ==> R') c d ->
  forall (a : A) (b : B), refines R a b -> refines R' (c a) (d b).
```

This rule is sufficient for dealing with any number of arguments thanks to currying. It is essential to break expressions into smaller bricks. For example, on the goal

```
refines ?R (2 + 3) ?e,
```

where `?R` and `?e` are existential variables denoting holes to be filled, the `refines_apply` rule leads to the two following subgoals

```
refines (?R ==> ?R) (fun n => 2 + n) ?f,
refines ?R 3 ?n,
```

thus partially instantiating `?e` as `?f ?n`. Note that we kept the same relation `?R` since `?f` must have identical input and output types.

A tactic is available to trigger type class inference. If one types `coqeal`, the whole goal is refined and then solved by computation (if CoqEAL contains refinements for all the elements of the goal). If one wants to refine only a part of the goal and perform no computation yet, it is possible to use the following lemma.

```
Lemma refines_eq (T : Type) (x y : T) : refines eq x y -> x = y.
```

Given `x`, using this lemma triggers type class inference to find an instance of

```
refines eq x ?y,
```

thus deriving the refinement `?y` of `x`. To be precise, `?y` is not exactly the refinement of `x`, since it has the same type. However, most refinement relations are defined using an equality that involves a function from the computation-oriented type to the proof-oriented type (recall `Rnat` in Section 8.1.1), so that `?y` is in fact such a function applied to the refinement of `x`.

In Section 8.2.3, we present a tactic that uses this idea of a function from the computation-oriented type to the proof-oriented in order to use refinement for the simplification of arithmetic expressions.

8.2.3 A Simplification Tactic

In the context of a proof such as the one in Section 3.2, computation is used to simplify arithmetic expressions. We designed a tactic in order to make it possible to easily use refinement for such computations.

The idea behind this tactic is simple: once we know a refinement of a given expression, we can compute efficiently on this new representation and, in particular, we can compute simplifications. Since this tactic is designed to simplify proof-oriented expressions, we use functions that go from computation-oriented types to proof-oriented types in order to get back proof-oriented expressions. Such functions must respect equality, i.e. they must be refinement of the identity function: we must be able to prove refinement theorems of the form:

```
Lemma refines_spec : refines (R ==> eq) id spec,
```

which unfolds to

```
forall P C, R P C -> id P = spec C.
```

Using such a theorem in combination with Lemma `refines_apply` and Lemma `refines_eq` makes it possible to design the following simplification strategy, illustrated by Figure 8.3: given a proof-oriented expression `P`,

1. trigger refinement inference using Lemma `refines_eq` on the term `id P`, so that a refinement `C` of `P` is inferred and `id P` is replaced with `spec C`.

2. use conversion on the computation-oriented object `C` to simplify it into `C'`.

3. use a reduction strategy of Coq (e.g. `simpl`, `compute` or `vm_compute`) in order to reduce `spec C'` to a simplified proof-oriented expression `P'`.

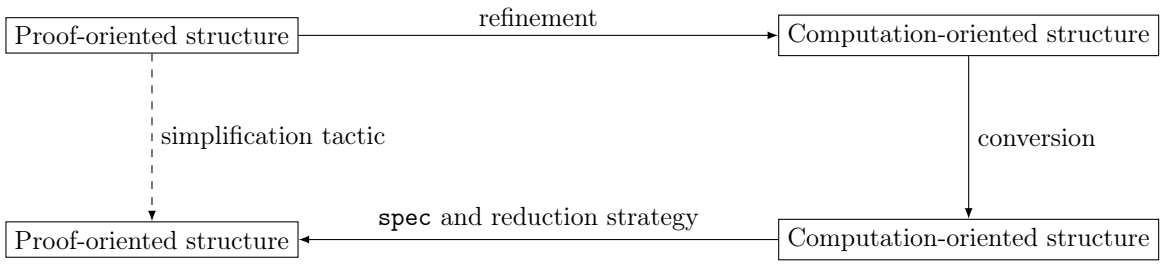


Figure 8.3 – Refinement-based Simplification Strategy

Let us give an example on polynomials. In Mathematical Components, a polynomial is implemented as the sequence of its coefficients with a proof that the last element of the

sequence is non-zero (it is the leading coefficient). One can refine such a structure by removing the proof. Thus, the polynomial

```
1%:P + 'X - (1%:P * 'X),
```

where `%:P` casts a constant into the corresponding constant polynomial, is refined into

```
spec ([:: 1%C] +%C [:: 0%C ; 1%C] -%C ([:: 1%C] *%C [:: 0%C; 1%C])),
```

where `+%C`, `-%C` and `*%C` respectively denote the coefficient-wise addition, opposite and multiplication of sequences on computation-oriented coefficients. The conversion step then transforms the sequence in argument of `spec` into

```
[:: 1%C ; 0%C].
```

An appropriate `spec` function then removes the null coefficient in order to return the polynomial `1%:P`.

In our published work [CR17b], the only reduction strategy that was available was `simpl` and our simplification tactic was called `coqeal_simpl`. Cyril Cohen later reworked its infrastructure in order to make it possible to also use the `compute`, `vm_compute` and `native_compute` tactics and so that different tactics and constructions are available:

— `coqeal_ strategy` simplifies the whole goal using the given reduction strategy.

— `coqeal [pattern] strategy` finds a sub-expression in the goal that matches the pattern and then simplifies it using the reduction strategy.

— `[coqeal strategy of x]` simplifies `x` into `x'` using the given strategy and outputs a proof of `x = x'`.

We used in particular this last construction to give a one-liner proof of Lemma `det_ctmat1` from the extension of Mathematical Components for real closed fields [Coh]. This lemma states that the matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

has determinant 2. The formal proof is the following.

```
Lemma det_ctmat1 : \det ctmat1 = 2.
Proof.
by do ?[rewrite (expand_det_row _ ord0) //=;
  rewrite ?(big_ord_recl,big_ord0) //= ?mxE //=;
    rewrite /cofactor /= ?(addn0, add0n, expr0, exprS);
      rewrite ?(mul1r,mulr1,mulN1r,mul0r,mul1r,addr0) /=;
        do ?rewrite [row' _ _]mx11_scalar det_scalar1 !mxE /=].
Qed.
```

It is a succession of rewriting rules that computes the determinant by expanding it along the first row. Thanks to our tactics, this can be shortened into

```
Definition det_ctmat1 := [coqeal vm_compute of \det ctmat1].
```

Note that it is a `Definition` and not a `Lemma`: we do not have to manually compute the determinant in order to provide a statement to be proven. This is not critical on this example but it would be a great improvement for bigger matrices with larger coefficients. Moreover, if we type

```
Check det_ctmat1,
```

Coq answers as expected

```
det_ctmat1 : \det ctmat1 = 2.
```

8.3 The Benefits of Parametricity

Parametricity [Wad89] is a reformulation of Reynolds' abstraction theorem [Rey83]. It is based on the idea that all inhabitants of a (closed, i.e. with no free variable) type share a property, expressed through a relational interpretation of the type.

We first explain this interpretation and state the parametricity theorem in Section 8.3.1. Then, we show in Section 8.3.2 how parametricity makes it possible to further automate refinement: this automation does not apply to refinement inference but to the proof of refinement theorems. Finally, we discuss a collaboration that we started with Cyril Cohen, Assia Mahboubi, Matthieu Sozeau and Nicolas Tabareau in order to generalise parametricity (see Section 8.3.3).

8.3.1 The Parametricity Theorem

Reynolds introduced a relational interpretation of types in his work about polymorphism. More precisely, we are interested in *parametric polymorphism*: a function or a data type is said to be polymorphic if and only if it can be implemented in such a way that its behaviour does not depend on the type of its parameters. For instance, both the `vector` data type and `append` function given in the introduction of this thesis are polymorphic with respect to the type `A` of the elements: the elements may well be integers, matrices or even functions from vectors to real numbers, the way vectors are built and the computational behaviour of the `append` function are the same in every case.

From a type point of view, polymorphism is captured by the polymorphic type $\forall X.\ A$, where X is a type variable and A is a type where X may appear as a free variable.

Reynolds' relational interpretation of types was first expressed in terms of a set-theoretic model of the polymorphic λ-calculus [Rey83], which in fact does not exist [Rey84]. Wadler however transposed this interpretation to another context where models actually exist [Wad89] and Reynolds later provided an interpretation in terms of categories [MR91]. In this section, for the sake of simplicity, we stick to the set-theoretic view to give the intuition of the abstraction theorem.

Let us denote by $[\![A]\!]$ the interpretation of the type A as a relation. We give here examples of interpretations for constant types, function types and polymorphic types.

— A constant type is interpreted as the identity relation on its elements. For instance,

$$[\![\texttt{int}]\!] = \{(\texttt{x}, \texttt{x}) \mid \texttt{x} \ : \ \ \texttt{int}\}\,.$$

— The interpretation of function types exactly corresponds to the relation we described in Section 8.1.2: two functions are related if and only if whenever their argument are related, their outputs are related too. In short:

$$[\![\texttt{A -> B}]\!] = [\![\texttt{A}]\!] \texttt{ ==> } [\![\texttt{B}]\!] .$$

— Elements of a polymorphic type $\forall X.\ A$ can be seen as (dependent) functions: given a type represented by X, they compute an element of $A(X)$. Since these functions are polymorphic, their behaviour does not depend on the type that will replace X.

Thus, the interpretation of $\forall X.\ A$ must encompass the interpretation of **any** function type with input type T and output type $A(T)$. In particular, it depends on the interpretation $[\![A]\!]$ where the interpretation $[\![X]\!]$ is left abstract: it may be any relation between two input types. Two dependent function are then related if they agree with the interpretation of A for any two related input types.

$$[\![\forall X.\ A]\!] = \{(f, g) \mid \forall S.\ \forall T.\ \forall R \subseteq S \times T.\ (f(S), g(T)) \in [\![A]\!]\,\{[\![X]\!] \leftarrow R\}\} .$$

Parametricity for A can then be expressed as follows: for all closed term t of type A, we have $(t, t) \in [\![A]\!]$. In a less naive context, terms also need to be interpreted and parametricity also holds for terms and types that contain free variables, so long as these variables respect the relations corresponding to their types. However, in CoqEAL we use parametricity only on closed terms so we will not state the theorem in its full generality. Instead, we state a slightly less general parametricity theorem in Theorem 8.1.

Theorem 8.1 (Parametricity)**.** *For all closed type* `A` *and all closed term* `t : A`, $[\![\texttt{t}]\!]$ *is a proof of* $(\texttt{t}, \texttt{t}) \in [\![\texttt{A}]\!]$.

The parametricity theorem is stated for a simple calculus, but it can be extended to more complex systems. To do so, it is sufficient to give an interpretation to the additional type constructors and to prove that they satisfy the relation given in Theorem 8.1. This has been done for dependent types: Bernardy et al. [BJP12] and Atkey et al. [AGJ14] proposed two different interpretations for dependent types. Keller and Lasson [KL12] provided an interpretation for the Calculus of Inductive Constructions and implemented a plug-in for Coq that, given a closed term, instantiates the parametricity theorem on it. We helped updating this plug-in for a new version of Coq and we now use it in CoqEAL for data refinement.



Remark

This plug-in only provides **instantiations** of the parametricity theorem: this is a *meta-theorem*, i.e. it cannot be proven inside the theory of Coq.

Some type theories were developed in order to internalise the parametricity theorem [BM13, BCM15], but to our knowledge no proof assistant based on such a theory exists.

8.3.2 Parametricity for Data Refinement

Wadler showed that the parametricity theorem can be used to derive "free theorems". Cohen et al. [CDM13] noticed that the compositionality of data refinement corresponded to these free theorems. In fact, the third step of the refinement methodology we described in Section 8.1.4 can be automated thanks to parametricity.

Indeed, the parametricity theorem gives properties about polymorphic functions, which are also called generic functions. The purpose of the first step of the refinement methodology (generic programming) is to obtain polymorphic functions for which the parametricity theorem will be relevant.

For instance, recall from Section 8.1.3 the definitions of the `seqmx` type and of its null element, `seqmx0`. They are polymorphic with respect to the type of the coefficients of matrices. Lemma `Rseqmx_0`, which we repeat here, corresponds to the second step of the methodology: proving that when coefficients are in a ring `R`, `seqmx0` indeed represents the null matrix of the type `'M[R]_(m,n)` of matrices on `R`.

```
Lemma Rseqmx_0 (R : ringType) (m n : nat) :
  Rseqmx (const_mx 0%R) (@seqmx0 R _ m n).
```

The last step of the methodology is to use the parametricity of `seqmx0` with respect to the type of coefficients to deduce Lemma `RmxC_0`, which we also repeat, from Lemma `Rseqmx_0`.

```
Lemma RmxC_0 (R : ringType) (C : Type) (rC : R -> C -> Type) ...
  (m n : nat) : RmxC rC (const_mx 0%R) (@seqmx0 C _ m n).
```

The relation `RmxC` is defined as the composition of the relation between `'M[R]_(m,n)` and `@seqmx R` and another relation that lifts a relation between `R` and a computation-oriented type `C` to matrices. This second relation uses a function `list_R`, which is in fact the relational interpretation of the (polymorphic) type of sequences: the type `seq` is interpreted as a function that takes a relation between two types `R` and `C` and that returns a relation between `seq R` and `seq C`.

```
Definition RmxC {R : ringType} {C : Type} (rC : R -> C -> Type)
  {m n : nat} : 'M[R]_(m,n) -> @seqmx C -> Type :=
  (Rseqmx \o (list_R (list_R rC)))%rel.
```

Forgetting for now its argument of type `zero_of A`, and assuming the dimensions `m` and `n` of the matrix are global variables and not arguments, `seqmx0` is of type `forall A, seqmx A`, or equivalently `forall A, seq (seq A)`. Theorem 8.1 thus proves that for any type `R` and `C` and any relation `rC` between `R` and `C`, `@seqmx0 R` is related to `@seqmx0 C` by the interpretation of `seq (seq A)` where `rC` replaces the interpretation of `A`. In other terms, the parametricity theorem proves "for free"

```
(list_R (list_R rC)) (@seqmx0 R) (@seqmx0 C).
```

Combining this "free theorem" with Lemma `Rseqmx_0`, we indeed deduce Lemma `RmxC_0`. If we take into account the other argument of `seqmx0`, we can notice that `seqmx0` is of type `forall A, A -> seq (seq A)`. Indeed, the `zero_of` type class only hides an element of `A`. Unfolding the interpretation of function types, the actual "free theorem" about `seqmx0` is the following:

```
rC 0%R 0%C -> (list_R (list_R rC)) (@seqmx0 R 0%R) (@seqmx0 C 0%C).
```

The additional assumption in this theorem corresponds to the omitted hypothesis of Lemma `RmxC_0`.



Remark

The dimensions of the matrix are actually considered as arguments by the function that computes the parametricity interpretation. This makes statements more complex, since the relational interpretation of `nat` appears. Fortunately, this relation coincides with the identity relation so that we can specialise theorems to remove the extra assumptions.

Ideally, one would indicate to the interpretation function the arguments with respect to which one wants to exploit parametricity. We hope that the work we describe in Section 8.3.3 will allow such modularity.

Thus, thanks to generic programming and to the parametricity plug-in implemented by Keller an Lasson, we get automated proofs for the compositionality of data refinement.

8.3.3 Current and Future Work on Parametricity

We saw in Section 8.3.2 that the parametricity theorem gives "for free" proofs of compositionality for data refinement. In fact, the abstraction theorem is really more general and captures a lot of different applications.

Depending on the context, one may add conditions on the relations that are used in the interpretation of polymorphic types. Sometimes, these conditions are necessary to prove the abstraction theorem. This is the case for instance in the work of Gilcher et al. [GLT17], who coined the expression *conditional parametricity*.

In other cases, these conditions define the kind of "free theorems" one may expect:

— When refinement relations are used, we obtain compositionality theorems for data refinement.

— Gross et al. [GEC18] noticed the importance of abstraction for reification (we introduce reification in Section 9.1) but their work can be generalised: we noticed that the abstraction theorem in fact proves the correctness of reification when it is used on the appropriate relation.

— Tabareau et al. [TTS18] require the relations to be equivalences (not equivalence relations but what is called equivalence in homotopy type theory [UFP13]) to define *univalent parametricity*.

Together with Cyril Cohen, Assia Mahboubi, Matthieu Sozeau and Nicolas Tabareau, we plan on building a framework that is general enough to share proofs between parametricity for refinement and univalent parametricity. We noticed that a reformulation of the property "being an equivalence" may factor both notions.

Indeed, we believe that refinement relations correspond to *functional relations*. The intuition is that for a functional relation R, for any x there exists exactly one y such that $(x, y) \in R$. This implies in particular that there is a function f such that

$$\forall x.\ \forall y.\ (x, y) \in R \Leftrightarrow y = fx.$$

Moreover, a relation R is an equivalence if and only if it is functional and its converse is also functional. With enough care for modularity, it should thus be possible to reuse the proof of the abstraction theorem for functional relations in the proof of this theorem for equivalences.



Remark

We give here a more precise explanation for the reader who is fluent in homotopy type theory.

This reformulation corresponds to the definition of equivalence based on contractible maps. In particular, a relation R is functional if and only if

$$\forall x.\ isContr\left(\sum_{y} R\ x\ y\right).$$

We also think that a proper instrumentation of the parametricity interpretation may help automatically finding the appropriate association list for proofs of correctness of reification (see Section 9.1 for the utility of association lists for reification). We plan to experiment with an implementation of the parametricity interpretation using MetaCoq [ABTS18, ABC+18, ACB+], which is simpler than the plug-in we use in CoqEAL.

Part III
Tools for Automation

CHAPTER 9

PROOF BY REFLECTION

Reflection [Bou97], also known as computational reflection [Har95], is a proof methodology based on computation. It has been used to implement several decision procedures in Coq, such as the `ring` [GM05] and `field` [DM01] tactics.

We discuss the original reflection methodology in Section 9.1 before describing a more modular methodology based on refinement, which we implemented in a prototype of tactic analogous to `ring` (see Section 9.2). Finally, we discuss in Section 9.3 possible improvements on this prototype and future work that will make it more general.

This chapter may be seen as an extended version of our publication on the topic [CR17b]. This work was done in collaboration with Cyril Cohen. The code snippets in Section 9.1 come from Coq's standard library [CDT19]. Those in Section 9.2 come from the CoqEAL library [CCD+].

9.1 Principles of Proof by Reflection

As mentioned in Section 8.2.1, it is possible to prove theorems through computation thanks to decision procedures. This does not apply to any theorem, since some problems are undecidable [Mon76]. However, there are properties that are decidable, i.e. there exists an algorithm that, given an object, decides whether or not this object has the aforesaid property.

When such an algorithm is implemented and proven correct in a proof assistant, it can serve as a basis for proof techniques. Indeed, if an algorithm `A` decides a predicate `P` on type `T`, its correctness theorem will take the following form:

```
Lemma A_correct (x : T) : P x <-> A x = true.
```

Thus, the algorithm `A` can be used directly in order to prove that `P` holds on some object `x`. It is sufficient to apply Lemma `A_correct` and then let computation do the proof: either `A`

returns `true` and the property is proven thanks to the reflexivity of equality or `A` returns `false` and we know that the property does not hold on `x`.

A semi-decision procedure is sometimes sufficient: if one already knows (without any formal proof) the expected output of `A`, it is sufficient to use this technique only on instances that succeeds. Thus, it is not necessary to know that `A` answers `true` on all `x` such that `P x`, i.e. that `A` is *complete* but that `P x` holds whenever `A` answers `true`, i.e. that `A` is *sound* (see Lemma `A_sound`).

```
Lemma A_sound (x : T) : A x = true -> P x.
```

However, for this proof technique to be efficient, it is necessary that the algorithm `A` both has a good computational complexity and manipulates a computation-oriented data structure. Hence, one often has to translate the goal to an appropriate data type.

For Boutin [Bou97], appropriate data types are inductive types since one can build efficient procedures based on pattern matching. Thus, the goal will be *reflected* to an abstract syntax tree, based on an inductive type, on which the (semi)-decision procedure will operate. This translation step is called *metaification* by Boutin, since it is usually not implemented in the language of terms but in a meta-language (e.g. the language $\mathcal{L}_{tac}$ [Del00] of tactics in Coq). Nowadays, the name *reification* is preferred.

The soundness theorem of the (semi-)decision procedure is now stated as follows:

```
Lemma A_sound (e : AST) : A e = true -> P (interp e),
```

where `interp` is an interpretation function that translates abstract syntax trees back to terms in `T`. This function may depend on an association list that maps variables to values that could not be reified in the tree (we will give an example later).

Hence, the reflection methodology (to prove `P x`) is the following 3-step strategy, illustrated by Figure 9.1:

1. Reify `x` into an abstract syntax tree `e`.
2. Compute `A e`.
3. Apply Lemma `A_sound` to conclude the proof.

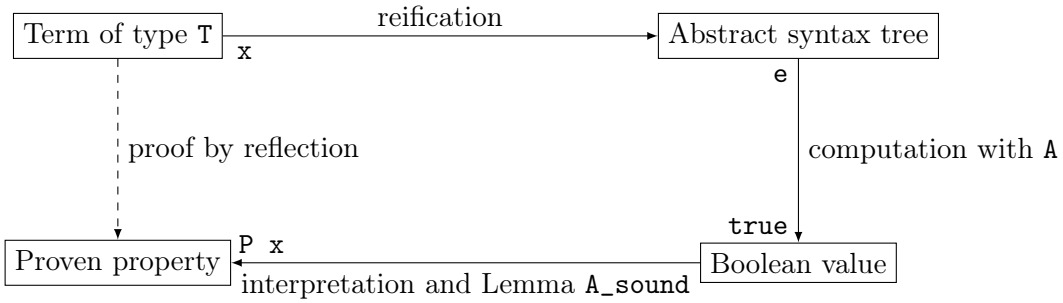


Figure 9.1 – The Reflection Methodology

The important point is that both the interpretation and the proof of the property are done through computation. Thus, reflection is not only an efficient way to prove theorems, but also to have short proofs. Indeed, in Coq, computation is done through conversion, which does not appear in proof terms. For instance, the term `eq_refl n`, i.e. the reflexivity of equality

instantiated on the integer `n : nat`, is a perfectly valid proof of `0 + n = n`, since both sides of the equation are convertible; the computation that transforms `0 + n` in `n` does not appear in the proof term. As a consequence, the proof of `P x` does not contain any trace of the translation from `x` to `interp e` nor of the step that transforms `A e` into `true`, since they are convertible: the proof term is only the application of Lemma `A_sound` to the reflexivity of equality.



Example

We illustrate this methodology with the example of the `ring` tactic [GM05], which decides equality between two arithmetic expressions up to the axioms of rings.

This tactics works as follows: both expressions are put in normal form through a procedure based on reflection and then compared. If the normal forms are equal, then the arithmetic expressions are equal. More precisely, the methodology illustrated by Figure 9.1 is implemented as follows:

1. Reify each expression into an abstract syntax tree representing polynomial expressions with integer coefficients.

This is done by observing the head operator in the expression and interpreting it as an operator over polynomials, and then recursively reifying its arguments. The abstract syntax tree data type provides constructors for the arithmetic expression generated by the following grammar:

$$e \quad ::= \quad 0 \mid 1 \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \mid -e \mid e^n \mid c,$$

where n ranges over natural numbers and c over ring-specific constants for which the user has provided an interpretation. For the remaining operations that have no interpretation, the sub-expression that cannot be reified is translated as an indeterminate and stored in an association list for interpretation.

For instance [CDT19], the expression

```
((f(5) + x) * x) + ((if b then 4 else f(3)) * 2)
```

is reified into the polynomial

```
((Y + Z) * Z) + (X * 2),
```

where the map `{X -> if b then 4 else f(3), Y -> f(5), Z -> x}` is stored. Here, `f` and `x` are variables and consequently have no interpretation as polynomial operations/objects and the `if` construct is not a standard ring operation, although it could be encoded as follows.

```
if b then x else y = x * b + y * (1 - b).
```

2. Normalise the abstract syntax trees into a sparse representation of Horner polynomials and compare the normal forms. Note that this implies another change of representation and that this is done by computation.

Horner's representation of polynomials is the following: the polynomial

$$a_0 + a_1 * X + a_2 * X^2 + \cdots + a_n * X^n$$

is represented as

$$a_0 + X * (a_1 + X * (a_2 + \cdots + X * a_n) \ldots)).$$

This representation can be generalised to multivariate polynomials.

Grégoire and Mahboubi implemented an algorithm `norm` that computes a normal form in this representation for polynomials and another one, `Peq`, that compares two normal forms and returns `true` if they are equal.

3. The soundness lemma for this procedure is stated as follows (we simplified the actual statement to hide implementation details for the sake of clarity):

```
Lemma ring_correct (e1 e2 : AST) (l : map) :
  Peq (norm e1) (norm e2) = true -> interp l e1 = interp l e2.
```

Note that the two abstract syntax trees must share the association list since a single sub-expression must be represented by the same indeterminate in both polynomials.

A simplification tactic, `ring_simplify`, based on this procedure, also exists. Its soundness lemma could be stated as follows (this statement is also simplified):

```
Lemma ring_rw_correct (e : AST) (P : HornerPoly) (l : map) :
  norm e = P -> interp l e = interp_Horner l P,
```

where `interp_Horner` evaluates a polynomial in Horner representation into a ring expression.

9.2 A More Modular Methodology

Decision procedures based on reflection are often designed in a monolithic fashion: they rely on ad-hoc data structures to which specific transformations are applied. To implement variations and/or improvements, one has to dive into the core of such procedures and change the structures/proofs. This may require the development of an extensive theory of the involved structures. For instance, in our example about the `ring` tactic in Section 9.1, two different data structures for polynomial expressions are used and it is necessary to link them in order to prove Lemma `ring_correct`.

We propose a more modular methodology for reflection, which makes a clear distinction between the proofs of soundness for decision procedures and the computations they will perform. Thanks to the use of refinement (see Section 8.1), we may use different data structures for the proof of soundness of a decision procedure and for its execution.

Our approach has two main benefits. On one hand, one can reuse the results of libraries such as Mathematical Components to prove soundness. This reduces the amount of lemmas to be proven. Moreover, such libraries usually contain structures that are well-suited for proofs, which means that the subtleties of the ad-hoc data structures used for computation will not impact the proof scripts. On the other hand, in this framework the soundness of each computable operation with respect to the proof-oriented one can be proven independently

from the others. Thus, not only soundness is easier to prove but adding/removing/changing an operation has less impact in terms of proofs than with the former methodology.

Our suggestion of strategy to use refinement is the following. We still want to prove that a property `P` holds on a term `x` by using a (semi-)decision procedure `A`. The idea is to implement `A` through generic programming so as to prove its soundness on a proof-oriented data structure and to use a computation-oriented data structure to actually execute it. The soundness theorem for `A` can be stated as follows:

```
Lemma A_sound (p : PO_type) : A p = true -> P (interp p),
```

where `PO_type` is the proof-oriented data type used for the proof of this theorem.

This leads to the following methodology, illustrated by Figure 9.2:

1. Reify `x` into a proof-oriented object `p`.
2. Refine `p` to an equivalent computation-oriented object `c`.
3. Execute `A` on `c`.
4. Deduce `A p = true` thanks to the parametricity of `A`.
5. Apply Lemma `A_sound` to conclude the proof.

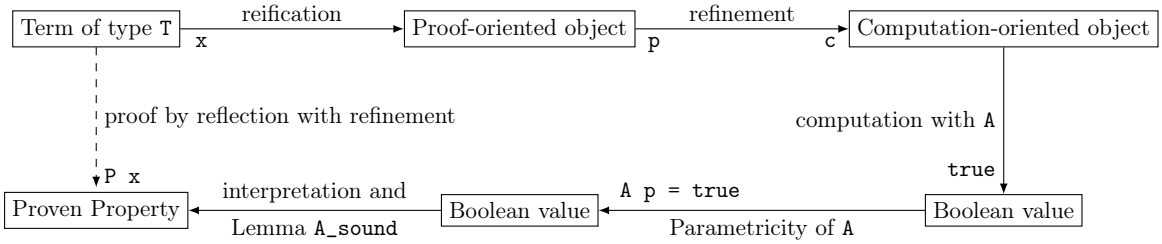


Figure 9.2 – The Methodology of Reflection with Refinement

Warning

The interpretation function is (in general) no longer computable since it manipulates a proof-oriented structure. As a consequence, the translation from `x` to `interp p` will appear in the proof term.

Refinement also leaves a trace in the proof term. In fact, we trade a part of the efficiency of reflection for a better modularity that eases the development and maintenance of tactics.

Example

We implemented a prototype of a tactic named `coqeal_ring`, which is available in CoqEAL [CCD+], and that tries to reproduce the `ring` tactic with this methodology. Actually, our implementation diverges from this methodology in a few places that we will pinpoint in our description.

1. We reify arithmetic expressions into the data type for polynomials in Mathematical Components [MCT].

Since Mathematical Components does not contain a structure for multivariate polynomials, we use an iterated structure of univariate polynomials, i.e. we replace $\mathbb{Z}[X_1, \ldots, X_n]$ with $\mathbb{Z}[X_1] \ldots [X_n]$. This implies the choice of an ordering of the variables, which must be the same for the two expressions.

This step, implemented as a tactic named `polyfication`, goes in fact through a reification into an abstract syntax tree that is then translated into a polynomial by a function `ast_to_poly`. The correctness of this function is stated as follows (we simplified the statement):

```
Lemma polyficationP (e : AST) (l : map) :
  interp l e = eval_poly l (ast_to_poly e),
```

where `eval_poly` generalises the polynomial evaluation function from Mathematical Components to iterated polynomials. **This function is not executable.**

After executing the `polyfication` tactic on the goal

```
e1 = e2,
```

the goal is

```
eval_poly l p1 = eval_poly l p2.
```

2. We refine the polynomials from Mathematical Components to a structure that implements Horner's representation of polynomials.

This structure is not exactly the one that is used in the implementation of `ring` but it already existed in CoqEAL. In particular, all the refinement theorems were already proven.

3. At this point, we deviate from the methodology. Although an equality test exists in CoqEAL for Horner polynomials, there is no normalisation function in this library. Instead of implementing such a function and proving its refinement theorem, we preferred fast prototyping over an actual implementation of the methodology so that we exploited another tool that was available: our simplification tactic described in Section 8.2.3.

Thus, this step corresponds to the second step in Figure 8.3 and we only apply conversion to simplify the polynomials in Horner representation.

4. This step now corresponds to the third step in Figure 8.3: we use the already proven correct `spec` function and the `vm_compute` reduction strategy to get back polynomials from Mathematical Components.

The three last steps are implemented in a tactic named `coqeal_vm_compute_eq2` and that uses twice our tactic from Section 8.2.3 to transform the goal

```
eval_poly l p1 = eval_poly l p2
```

into

```
eval_poly l q1 = eval_poly l q2,
```

where q_1 (respectively q_2) is the simplification of p_1 (respectively q_2).

5. We use a set of rewriting rules in order to evaluate the polynomials on the association list and get back arithmetic expressions. This set of rules is applied by a tactic called `depolyfication`.

The `coqeal_ring` tactic successively applies the three tactics we just described, which is analogous to applying the `ring_simplify` tactic on both sides of the equation, and then attempts to close the goal by reflexivity. This strategy is depicted in Figure 9.3.

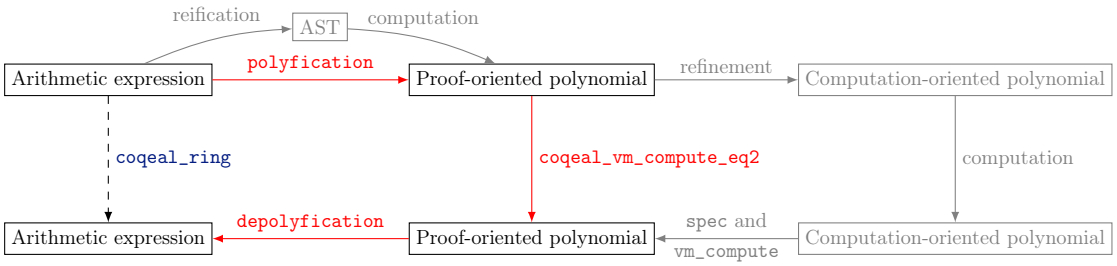


Figure 9.3 – The `coqeal_ring` Tactic

9.3 Possible Improvements and Future Work

Our prototype of tactic to prove equations in rings may be improved in several ways: it is missing some features of the `ring` tactic (see Section 9.3.1), its present implementation is inefficient (see Section 9.3.2) and it opens the door to new use cases at a lesser implementation cost than with the `ring` tactic thanks to the modularity of our methodology (see Section 9.3.3).

9.3.1 Missing Features

Presently, the implementation of `coqeal_ring` is incomplete. CoqEAL is missing some refinements, like the power function and some coercions. Although the `polyfication` tactic takes them into account, it is not possible to simplify the obtained polynomials through the `coqeal_vm_compute_eq2` tactic since they cannot be refined.

Moreover, the reification step of `ring` takes into account ring-specific constants for which the user defined a translation. This is not the case of `polyfication`.

Another missing feature of our prototype is the possibility to use other rings than the set of integers as the ring of coefficients for polynomials. The ring of integers is a natural choice since there is a canonical injection from integers to any ring (in terms of category theory, we say that the ring of integers is an initial object of the category of rings). However, it may happen that another ring (e.g. $\mathbb{Z}_{/n\mathbb{Z}}$ or rational numbers) is a better choice. For instance, the equation

$$a + a = 0$$

is true for any a in the ring of boolean values. This is not provable if we use integers as coefficients for the polynomial representation ($2X$ and 0 are two different normal forms) but it is if we use boolean values instead. Indeed, $a + a$ is represented by the polynomial

$$X + X = (1 + 1) * X = 0 * X = 0,$$

which is the same normal form as for the right-hand side of the equation.

Finally, Lemma `polyficationP` holds only for commutative rings, while the `ring` tactic can also be used on commutative semi-rings.

9.3.2 Efficiency issues

The `coqeal_ring` tactic suffers from efficiency issues that we mainly attribute to the refinement step. Indeed, the bottleneck of our current prototype is the refinement of polynomial operations which looks exponential in the number of indeterminates. This is caused by the fact that we are using a nested data structure (iterated univariate polynomials), which implies a lot of backtracking in type class resolution.

Cyril Cohen tried to improve the way backtracking in type class resolution works in order to make refinement linear in the number of indeterminates but did not succeed because of the complexity of the implementation. Another solution, which is currently under experimentation, is to replace iterated univariate polynomials by the structure of multivariate polynomials developed by Strub [BBRS16, Str]. A refinement of this structure, now integrated into CoqEAL, was developed by Martin-Dorel and Roux for the design of a reflexive tactic to decide polynomial positivity [MR17]. We are currently extending this refinement in order to use it in the `coqeal_ring` tactic.

Another efficiency issue may appear in the `depolyfication` tactic. As of today, it is instantaneous on our small examples, but we expect that for bigger terms it could become slower because it is based on rewriting. Indeed, rewriting involves unification and, in spite of optimisations on the `rewrite` tactic in SSReflect in order to lower the number of considered non-solvable unification goals, this method is still expensive in terms of running time.

We suggest to replace this step either with the same process as for the `polyfication` tactic (going through an abstract syntax tree to replace rewriting by computation), or with a refinement of polynomial evaluation. The latter would remove the need for `depolyfication` since the simplification step based on refinement would directly evaluate back the polynomials into arithmetic expressions.

9.3.3 Possible Generalisations

Once our tactic's proof power catches up with `ring` and the efficiency issues are solved, it will be possible to further improve its proof power thanks to the modularity of our method.

A slight modification of the `polyfication` tactic would allow us to handle morphisms, making it possible to translate the expression $f(x+y)-f(y)$, where f is a morphism, into $X+Y-Y$ with the variable map $[f(x); f(y)]$ and thus to simplify it into $f(x)$. Indeed, the Mathematical Components library contains canonical structures for morphisms. This inference mechanism would then allow us to automatically recognise morphisms during reification.

We could also bring more flexibility to the reduction strategy. Instead of using the reduction strategy from Section 8.2.3, we suggest the user could plug in his own transformation, like root finding or factoring, which would make it possible to go from an equation of the form $x^2 + 2bx + c = 0$ to the conjunction of $b^2 - c \geqslant 0$ and $x = -b \pm \sqrt{b^2 - c}$. One can even imagine plugging in here an external tool which produces a proof witness.

Finally, another possible improvement would be the use of Gröbner bases [Buc65] to reason modulo equations. The current `ring` tactic already deals with hypotheses of the form $m = p$ where m is a monomial and p a polynomial, but thanks to Gröbner bases m could be any polynomial. For instance, this would help us in the proof of stability for the inverted pendulum (given in Section 3.2), where we extensively use the equation $p_2^2 + p_3^2 = 1$.

Buchberger's algorithm for computing Gröbner bases was formalised by Théry [Thé01]. Pottier and Théry used Coq's extraction mechanism to obtain a program from this formalisation, called gbcoq [PT98]. Pottier later used gbcoq, together with other programs computing Gröbner bases, to build a reflexive tactic `gbR` proving equations in rings in a sceptical way [Pot08] but which is not fully automated: the `gbR` tactic calls gbcoq, which builds a certificate, and outputs a small proof script based on this certificate, which is meant to close the goal if it replaces the line that calls `gbR`.

We rather suggest a fully automated autarkic approach by integrating the formalised algorithm into our tactic and running it inside Coq thanks to refinement. Implementing this approach is made easier by Théry's adaptation of his formalisation to the Mathematical Components library [Thé], based on Strub's formalisation of multivariate polynomials, for which a refinement is already available.

CONCLUSION

Assessment of our Contributions

The purpose of this thesis was to evaluate available tools for classical analysis in Coq and to propose solutions to their weaknesses in order to make interactive theorem proving in mathematics easier. The best way of acquiring experience on a tool being to use it in concrete situations, this thesis revolves around a case study in control theory.

I first used the Coquelicot library to formalise a proof of stability for the inverted pendulum. This proof of correctness of a control function for this non-linear dynamical system required two steps: first the proof of LaSalle's invariance principle, a standard theorem in stability analysis, and then its application to the case of the inverted pendulum. During both steps, I not only accomplished the formal verification of the mathematical proof, but I also worked on this proof, either to generalise the result that is proven or to correct errors in the proof. In an ideal world, such a proof would be directly developed in a proof assistant so that errors would be spotted at the time of making them and that no false result could be considered proven.

The proof of LaSalle's invariance principle using Coquelicot allowed me to learn to use filters, which are not intuitive for someone who has done proofs using $\varepsilon - \delta$ definitions for years. However, my experimentation with filters once again confirmed how convenient they are to prove results about convergence. In particular, I formalised topological notions such as compactness using filters, which proved to be particularly appropriate for LaSalle's invariance principle. This proof was moreover the occasion to introduce notations based on a filter inference mechanism in order to write theorems that look closer to their pen-and-paper equivalents.

The application of our formalisation of LaSalle's invariance principle to the inverted pendulum also allowed for the design of new tools that bridge the gap between formal and pen-and-paper proofs. Indeed, our inference mechanism for the computation of differentials and derivatives frees the user from such computation, which is often non-visible in proofs. Moreover, our connection of Coquelicot with Mathematical Components gives access to all the facilities of Mathematical Components for linear algebra, which will be a great help for multivariate analysis.

In spite of these new tools, there remained aspects of formal proofs with Coquelicot that are unnatural for a mathematician, even for one who is now fluent in filter manipulation, and there were still issues due to my use of Coquelicot in a context it was not designed for (e.g. classical reasoning, combination with Mathematical Components). Hence, I started a collaboration with other researchers in order to develop a new library that would palliate these issues. This library integrates the tools I developed for the case study and extends a subset of Coquelicot with new theories, especially in topology, and with tools for asymptotic reasoning. Our set of tactics for small-scale filter elimination in particular allows for an intermediate between $\varepsilon - \delta$ definitions and their filter-based equivalent, making statements closer to those expressed using $\varepsilon - \delta$ definitions while still keeping the abstraction provided by filters. Moreover, they make it possible to write proofs as if they were informal, manipulating "near enough" values, which is also the purpose of our implementation of Bachmann-Landau notations.

Although we made progress towards an easier formalisation of mathematics thanks to a better collaboration between libraries and to tools to make formal proofs closer to pen-and-paper ones, new challenges emerged. In particular, the `near` tactics are not intuitive enough, which means that we need to improve them so that they may contribute to lower the level of expertise in formal proofs required to formalise mathematics in Coq. Moreover, it is harder to do proofs by large-scale reflection than with Coquelicot, which is also the case for the Mathematical Components library.

I started a project on proof by computation to address this issue. Thanks to the introduction of refinement in the reflection methodology, I obtained a more modular methodology that eases the proof of correctness of a reflection-based decision procedure while computation on efficient data structures is still possible. Moreover, this new methodology is flexible enough to allow for generalisations of existing tactics, thus widening their domain of application, which means that more proofs could be automated.

Perspectives

The new reflection methodology based on refinement seems promising. It opens the door to a more efficient development of reflexive tactics thanks to its modularity and the maximisation of code reuse. In spite of its early development stage, our prototype of reflexive tactic for arithmetic reasoning in rings paves the way towards generalisations of the `ring` tactic that would require more efforts on the present implementation of this tactic. Moreover, the use of refinement by Martin-Dorel and Roux in a reflexive tactic [MR17], although it is only applied to the verification procedure of a sceptical implementation, proves that the current inefficiencies of the refinement framework are not prohibitive. Our future work on parametricity should furthermore help improving the refinement framework. It would also be interesting to apply this methodology in other contexts that are not necessarily related to arithmetic.

Working on the Mathematical Components Analysis library is a great source of insight into the formalisation of mathematics. I learnt how important the choice of foundations is for the proof practice. Our choice of strong axioms definitely eased the development of tools that make formal proofs closer to pen-and-paper ones. Some of my collaborators are currently investigating to determine if weaker axioms can be used without damaging the proof style. This is all the more important as we will share insights on the formalisation of integrals with

the members of the MILC project [ABC+], who use Coquelicot with axioms that are similar to ours to develop a formalisation of the Lebesgue integral [Leb50].

Beyond analysis, the Mathematical Components Analysis library is a step towards a unified framework for formal mathematics in Coq. Its compatibility by design with Mathematical Components turns the combination of both libraries into a significant data base of theories ranging from general algebra to analysis by way of topology and graph theory. In particular, this should ease the formalisation of theories that involve both linear algebra and analysis such as for instance the discrete Fourier transform and its applications (e.g. partial differential equation resolution or polynomial multiplication).

Finally, this work also revealed that developing powerful tools that ease the proof process and that bridge the gap between formal and pen-and-paper proofs is not sufficient. More precisely, these tools may become a hindrance to the use of proof assistants for non-experts if they are insufficiently intuitive. A great effort has to be put on documentation and, most of all, on communication with users to understand and to try to meet their need. This is a necessary condition to spread the use of formal methods.

LIST OF FIGURES

1.1 The Two Equilibria of the Free Pendulum 4
1.2 The Inverted Pendulum 4
1.3 Two Kinds of Control 5
1.4 Closed-Loop Control for the Inverted Pendulum 5
1.5 The Inverted Pendulum with Annotations 9
1.6 Illustration of the Notions of Stability 10
1.7 The Computed Control Function 12
1.8 Steps in the Formal Study of a Physical System 14

2.1 A Vector Field 16
2.2 Contour Map of a Lyapunov Function 17
2.3 Illustration of Convergence to a Set 18
2.4 Illustration of the Original Invariance Principle 19
2.5 Illustration of the Stronger Invariance Principle 23
2.6 The Coquelicot Hierarchy 25
2.7 Different Kinds of Neighbourhoods 28
2.8 Illustration of T_2 Separation 34

3.1 The Inverted Pendulum with Annotations (repeated) 44

5.1 The Coquelicot Hierarchy (repeated) 75
5.2 Partial Hierarchy of Mathematical Components Analysis 78
5.3 Hierarchy of Mathematical Components Analysis 85

8.1 The Believing Approach 121
8.2 The Sceptical Approach with Certificates 122
8.3 Refinement-based Simplification Strategy 125

9.1 The Reflection Methodology 134
9.2 The Methodology of Reflection with Refinement 137
9.3 The `coqeal_ring` Tactic 139

BIBLIOGRAPHY

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017.

[ABC+] Stéphane Aubry, Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. MILC: Mesure et Intégrale de Lebesgue en Coq. https://lipn.univ-paris13.fr/MILC/index.php.

[ABC+18] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In Avigad and Mahboubi [AM18], pages 20–39.

[ABTS18] Abhishek Anand, Simon Boulier, Nicolas Tabareau, and Matthieu Sozeau. Typed Template Coq – Certified Meta-Programming in Coq. In *CoqPL 2018 - The Fourth International Workshop on Coq for Programming Languages*, pages 1–2, Los Angeles, CA, United States, January 2018.

[ACB+] Abhishek Anand, Cyril Cohen, Simon Boulier, Gregory Malecha, Matthieu Sozeau, and Nicolas Tabareau. metacoq: Metaprogramming in Coq. https://github.com/MetaCoq/metacoq.

[ACM+] Reynald Affeldt, Cyril Cohen, Assia Mahboubi, Damien Rouhling, and Pierre-Yves Strub. The Mathematical Components Analysis Library. https://github.com/math-comp/analysis/releases/tag/0.2.2. Version 0.2.2.

[ACM+18] Reynald Affeldt, Cyril Cohen, Assia Mahboubi, Damien Rouhling, and Pierre-Yves Strub. Classical Analysis with Coq. In Sozeau and Tabareau [ST18]. 2-pages abstract.

[ACR18] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *Journal of Formalized Reasoning*, 11(1):43–76, 2018.

[AD04] Jeremy Avigad and Kevin Donnelly. Formalizing O Notation in Isabelle/HOL. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2004.

[AdMK18] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Available at https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf, Nov 2018. Release 3.4.0.

[AE09] Alessandro Arsie and Christian Ebenbauer. Refining LaSalle's Invariance Principle. In *Proceedings of the 2009 Conference on American Control Conference*, ACC'09, pages 108–112, Piscataway, NJ, USA, 2009. IEEE Press.

[AF88] Jean-Marie Arnaudiès and Henri Fraysse. *Cours de mathématiques*, volume 2, Analyse. Dunod, 1988.

[ÅF00] Karl Johan Åström and Katsuhisa Furuta. Swinging up a pendulum by energy control. *Automatica*, 36(2):287–295, 2000.

[AF18] June Andronick and Amy P. Felty, editors. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. ACM, 2018.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jouannaud and Shao [JS11], pages 135–150.

[AGJ14] Robert Atkey, Neil Ghani, and Patricia Johann. A Relationally Parametric Model of Dependent Type Theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014.

[AK15] Abhishek Anand and Ross A. Knepper. ROSCoq: Robots Powered by Constructive Reals. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2015.

[ÅM08] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, USA, 2008.

[AM17] Mauricio Ayala-Rincón and César A. Muñoz, editors. *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*. Springer, 2017.

[AM18] Jeremy Avigad and Assia Mahboubi, editors. *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*. Springer, 2018.

[Art04] Rob Arthan. The Eudoxus Real Numbers. Available at https://arxiv.org/abs/math/0405454, 2004.

[Bac94] Paul Bachmann. *Die Analytische Zahlentheorie*. B.G. Teubner, 1894.

[Bar68] Erwin H. Bareiss. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of Computation*, 22(103):565–578, 1968.

[Bar14] Itzhak Barkana. Defending the beauty of the Invariance Principle. *International Journal of Control*, 87(1):186–206, 2014.

[Bar17] Itzhak Barkana. Can Stability Analysis be really simplified? (Revisiting Lyapunov, Barbalat, LaSalle and all that). *AIP Conference Proceedings*, 1798(1):020017, 2017.

[BB02] Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.

[BBG+18] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar. *Journal of Automated Reasoning*, 61(1-4):9–32, 2018.

[BBRS16] Sophie Bernard, Yves Bertot, Laurence Rideau, and Pierre-Yves Strub. Formal Proofs of Transcendence for *e* and *pi* as an Application of Multivariate and Symmetric Polynomials. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 76–87. ACM, 2016.

[BC01] 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(1/2):3–22, 2001.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BCF+13] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[BCF+17] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formal proof of the Lax-Milgram theorem. In Bertot and Vafeiadis [BV17], pages 79–89.

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A Presheaf Model of Parametric Type Theory. *Electronic Notes in Theoretical Computer Science*, 319:67–82, 2015.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full Reduction at Full Throttle. In Jouannaud and Shao [JS11], pages 362–377.

[Ben06] Nick Benton. Machine Obstructed Proof - How many months can it take to verify 30 assembly instructions? In *1st Informal ACM SIGPLAN Workshop on Mechanizing Metatheory, Portland, Oregon, USA*, 2006.

[Ben13] Nick Benton. The Proof Assistant as an Integrated Development Environment. In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 307–314. Springer, 2013.

[Ber] Sophie Bernard. Fichiers de structures pour $\mathbb{R}$ et $\mathbb{C}$. https://github.com/Sobernard/Struct.

[Ber17] Sophie Bernard. Formalization of the Lindemann-Weierstrass Theorem. In Ayala-Rincón and Muñoz [AM17], pages 65–80.

[BF12] Lennart Beringer and Amy P. Felty, editors. *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*. Springer, 2012.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*, volume 5625 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2009.

[BG94] Leonid M. Brekhovskikh and Valery Goncharov. *Mechanics of Continua and Wave Dynamics*. Springer Series on Wave Phenomena. Springer-Verlag Berlin Heidelberg, second edition, 1994.

[BGBP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Paşca. Canonical big operators. In Mohamed et al. [MMT08], pages 86–101.

[BHMN15] Jasmin Christian Blanchette, Maximilian P. L. Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the Archive of Formal Proofs. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015.

[BJMD+] Nicolas Brisebarre, Mioara Joldes, Erik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Paşca, Laurence Rideau, , and Laurent Théry. The CoqApprox Library. http://tamadi.gforge.inria.fr/CoqApprox/.

[BJP12] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free - Parametricity for dependent types. *Journal of Functional Programming*, 22(2):107–152, 2012.

[BLM12] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2012.

[BLM15] 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.

[BLM16] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of Real Analysis: A Survey of Proof Assistants and Libraries. *Mathematical Structures in Computer Science*, 26(7):1196–1233, 2016.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011.

[BM13] Jean-Philippe Bernardy and Guilhem Moulin. Type-Theory In Color. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 61–72. ACM, 2013.

[BM17] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press - Elsevier, December 2017.

[BNUW09] Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors. *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[Bou71] Nicolas Bourbaki. *Topologie générale, Chapitres 1 à 4*. Éléments de mathématiques. Springer-Verlag Berlin Heidelberg, 1971.

[Bou74] Nicolas Bourbaki. *Topologie générale, Chapitres 5 à 10*. Éléments de mathématiques. Springer-Verlag Berlin Heidelberg, 1974.

[Bou97] Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997.

[BPP13] Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors. *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*. Springer, 2013.

[BRT18] Yves Bertot, Laurence Rideau, and Laurent Théry. Distant Decimals of π : Formal Proofs of Some Algorithms Computing Them and Guarantees of Exact Computation. *Journal of Automated Reasoning*, 61(1-4):33–71, 2018.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Leopold-Franzens-Universität Innsbruck, 1965.

[BV17] Yves Bertot and Viktor Vafeiadis, editors. *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. ACM, 2017.

[Caj28] Florian Cajori. *A History of Mathematical Notations*. The Open Court Publishing Company, Chicago, IL, 1928.

[Can14] Guillaume Cano. *Interaction entre algèbre linéaire et analyse en formalisation des mathématiques. (Interaction between linear algebra and analysis in formal mathematics)*. PhD thesis, University of Nice Sophia Antipolis, France, 2014.

[Car37a] Henri Cartan. Filtres et ultrafiltres. In *Comptes rendus hebdomadaires des séances de l'Académie des sciences* [MM.37], pages 777–779.

[Car37b] Henri Cartan. Théorie des filtres. In *Comptes rendus hebdomadaires des séances de l'Académie des sciences* [MM.37], pages 595–598.

[CCD+] Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg, Damien Rouhling, and Vincent Siles. The CoqEAL Library. https://github.com/CoqEAL/CoqEAL/releases/tag/1.0.0. Version 1.0.0.

[CD17] Raphaël Cauderlier and Catherine Dubois. FoCaLiZe and Dedukti to the Rescue for Proof Interoperability. In Ayala-Rincón and Muñoz [AM17], pages 131–147.

[CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[CDT19] The Coq Development Team. *The Coq proof assistant reference manual*, 2019. Version 8.9.1.

[CGW04] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21, 2004, Proceedings*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2004.

[CH88] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[CLH99] VijaySekhar Chellaboina, Alexander Leonessa, and Wassim M. Haddad. Generalized Lyapunov and invariant set theorems for nonlinear dynamical systems. *Systems & Control Letters*, 38(4–5):289 – 295, 1999.

[CLH+05] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005.

[CMST14] Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi. A Computer-Algebra-Based Formal Proof of the Irrationality of $\zeta(3)$. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2014.

[Coh] Cyril Cohen. real-closed: Theorems for Real Closed Fields in Mathematical Components. https://github.com/math-comp/real-closed.

[Coh12] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory.* PhD thesis, École polytechnique, Nov 2012.

[Coq85] Thierry Coquand. *Une théorie des constructions.* PhD thesis, Université Paris 7, janvier 1985.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[CPJ02] Debasish Chatterjee, Amit Patra, and Harish K. Joglekar. Swing-up and stabilization of a cart-pendulum system under restricted cart track length. *Systems & Control Letters*, 47(4):355–364, 2002.

[CRa] Cyril Cohen and Damien Rouhling. LaSalle: A formal proof of LaSalle's invariance principle. https://github.com/drouhling/LaSalle/releases/tag/1.0.0. Version 1.0.0, based on Coquelicot 3.0.2.

[CRb] Cyril Cohen and Damien Rouhling. LaSalle: A formal proof of LaSalle's invariance principle. https://github.com/drouhling/LaSalle/releases/tag/2.0.0. Version 2.0.0, based on Mathematical Components Analysis 0.2.2.

[CR17a] Cyril Cohen and Damien Rouhling. A Formal Proof in Coq of LaSalle's Invariance Principle. In Ayala-Rincón and Muñoz [AM17], pages 148–163.

[CR17b] Cyril Cohen and Damien Rouhling. A refinement-based approach to large scale reflection for algebra. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[CRLM16] Matthew Chan, Daniel Ricketts, Sorin Lerner, and Gregory Malecha. Formal Verification of Stability Properties of Cyber-physical Systems. In *CoqPL'16*, Jan 2016.

[CZČ05] Guanrong Chen, Jin Zhou, and Sergej Čelikovský. On LaSalle's invariance principle and its application to robust synchronization of general vector Liénard equations. *IEEE Transactions on Automatic Control*, 50(6):869–874, 2005.

[Dar90] Agata Darmochwał. Compact Spaces. *Formalized Mathematics*, 1(**2**):383–386, 1990.

[Del00] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[Dia75] Radu Diaconescu. Axiom of Choice and Complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dja18] Boris Djalal. *Formalisations en Coq pour la décision de problèmes en géométrie algébrique réelle*. PhD thesis, Université Côte d'Azur, 2018.

[DM01] David Delahaye and Micaela Mayero. Field, une procédure de décision pour les nombres réels en Coq. In Pierre Castéran, editor, *Journées francophones des langages applicatifs (JFLA'01), Pontarlier, France, Janvier, 2001*, Collection Didactique, pages 33–48. INRIA, 2001.

[DM05] David Delahaye and Micaela Mayero. Dealing with Algebraic Expressions over a Field in Coq using Maple. *Journal of Symbolic Computation*, 39(5):569–592, 2005.

[DM06] David Delahaye and Micaela Mayero. Quantifier Elimination over Algebraically Closed Fields in a Proof Assistant using a Computer Algebra System. *Electronic Notes in Theoretical Computer Science*, 151(1):57–73, 2006.

[DM10] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):2:1–2:20, 2010.

[dMKA+15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Felty and Middeldorp [FM15], pages 378–388.

[DMS12] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A Refinement-Based Approach to Computational Algebra in Coq. In Beringer and Felty [BF12], pages 83–98.

[DPGC15] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700. ACM, 2015.

[DSK98] Edward Doskocz, Yuri Shtessel, and Constantine Katsinis. MIMO Sliding Mode Control of a Robotic "Pick and Place" System Modeled as an Inverted Pendulum on a Moving Cart. In *Proceedings of Thirtieth Southeastern Symposium on System Theory*, pages 379–383, Mar 1998.

[Ebe17] Manuel Eberl. Proving Divide and Conquer Complexities in Isabelle/HOL. *Journal of Automated Reasoning*, 58(4):483–508, 2017.

[Ebe19a] Manuel Eberl. Verified Real Asymptotics in Isabelle/HOL. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '19, New York, NY, USA, 2019. ACM.

[Ebe19b] Manuel Eberl. Verified Solving and Asymptotics of Linear Recurrences. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 27–37. ACM, 2019.

[FKD13] Nicholas R. Fischer, Rushikesh Kamalapurkar, and Warren E. Dixon. LaSalle-Yoshizawa Corollaries for Nonsmooth Systems. *IEEE Transactions on Automatic Control*, 58(9):2333–2338, 2013.

[FM15] Amy P. Felty and Aart Middeldorp, editors. *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*. Springer, 2015.

[FMM+06] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[FMQ+15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In Felty and Middeldorp [FM15], pages 527–538.

[FR98] Michael J. Fischer and Michael O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 122–135, Vienna, 1998. Springer Vienna.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Paşca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Blazy et al. [BPP13], pages 163–179.

[Gar11] François Garillot. *Generic Proof Tools and Finite Group Theory. (Outils génériques de preuve et théorie des groupes finis)*. PhD thesis, École Polytechnique, Palaiseau, France, 2011.

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018.

[GEC18] Jason Gross, Andres Erbsen, and Adam Chlipala. Reification by Parametricity - Fast Setup for Proof by Reflection, in Two Lines of Ltac. In Avigad and Mahboubi [AM18], pages 289–305.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Berghofer et al. [BNUW09], pages 327–342.

[GJCP19] Armaë Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In ITP [ITP19]. Accepted paper.

[GL02] Benjamin Grégoire and Xavier Leroy. A Compiled Implementation of Strong Reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 235–246. ACM, 2002.

[Glo09] Stéphane Glondu. Extraction certifiée dans Coq-en-Coq. In Alan Schmitt, editor, *JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31 - February 3, 2009. Proceedings*, volume 7.2 of *Studia Informatica Universalis*, pages 383–410, 2009.

[Glo12] Stéphane Glondu. *Vers une certification de l'extraction de Coq. (Towards certification of the extraction of Coq)*. PhD thesis, Paris Diderot University, France, 2012.

[GLT17] Jan Gilcher, Andreas Lochbihler, and Dmitriy Traytel. Conditional Parametricity in Isabelle/HOL. In *TABLEAUX - Frontiers of Combining Systems (FroCoS) - Interactive Theorem Proving (ITP) 2017, Brasília, Brazil, September 26-29, 2017, Poster session*, 2017. Extended abstract.

[GM05] Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In Hurd and Melham [HM05], pages 98–113.

[GMT15] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[GN00] Herman Geuvers and Milad Niqui. Constructive Reals in Coq: Axioms and Categoricity. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2000.

[GNSW07] Herman Geuvers, Milad Niqui, Bas Spitters, and Freek Wiedijk. Constructive analysis, types and exact real numbers. *Mathematical Structures in Computer Science*, 17(1):3–36, 2007.

[Gué18] Armaël Guéneau. Procrastination - A proof engineering technique. In Sozeau and Tabareau [ST18]. 2-pages abstract.

[GZND11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to Make Ad Hoc Proof Automation Less Ad Hoc. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 163–175. ACM, 2011.

[Har95] John Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.

[Har05] John Harrison. A HOL Theory of Euclidean Space. In Hurd and Melham [HM05], pages 114–129.

[Har13] John Harrison. The HOL Light Theory of Euclidean Space. *Journal of Automated Reasoning*, 50(2):173–190, 2013.

[Har16] John Harrison. *The HOL Light System REFERENCE*, 2016. For 2016/10/19 revision.

[HIH13] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In Blazy et al. [BPP13], pages 279–294.

[Hil22] David Hilbert. Die logischen Grundlagen der Mathematik. *Mathematische Annalen*, 88(1):151–165, Mar 1922.

[HJO+12] Heber Herencia-Zapana, Romain Jobredeaux, Sam Owre, Pierre-Loïc Garoche, Eric Feron, Gilberto Pérez, and Pablo Ascariz. PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2012.

[HM05] Joe Hurd and Thomas F. Melham, editors. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*. Springer, 2005.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa72] Charles Antony Richard Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[How80] William A. Howard. The formulæ-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HW06] Florian Haftmann and Makarius Wenzel. Constructive Type Classes in Isabelle. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2006.

[IH12] Fabian Immler and Johannes Hölzl. Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL. In Beringer and Felty [BF12], pages 377–392.

[Imm18] Fabian Immler. A Verified ODE Solver and the Lorenz Attractor. *Journal of Automated Reasoning*, 61(1-4):73–111, 2018.

[IT16] Fabian Immler and Christoph Traut. The Flow of ODEs. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2016.

[IT19] Fabian Immler and Christoph Traut. The Flow of ODEs: Formalization of Variational Equation and Poincaré Map. *Journal of Automated Reasoning*, 62(2):215–236, 2019.

[ITP19] *ITP 2019 - Tenth International Conference on Interactive Theorem Proving*, Portland, United States, September 2019.

[JS11] Jean-Pierre Jouannaud and Zhong Shao, editors. *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*. Springer, 2011.

[Kel13] Chantal Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers. (Question de confiance : communication sceptique entre Coq et des prouveurs externes)*. PhD thesis, École Polytechnique, Palaiseau, France, 2013.

[Kha02] Hassan K. Khalil. *Nonlinear Systems*. Pearson Education. Prentice Hall, 2002.

[KL12] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[KLB+17] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the Gap – The Formally Verified Optimizing Compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*, Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium, pages 163–180, Bristol, United Kingdom, February 2017. CreateSpace.

[Knu98] Donald E. Knuth. Teach Calculus with Big O. *Notices of the AMS*, 45(6):687–688, 1998. Letter to the editor of the Notices of the American Mathematical Society.

[KS11] Robbert Krebbers and Bas Spitters. Type Classes for Efficient Exact Real Arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011.

[KW10] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2010.

[Lag11] Joseph-Louis Lagrange. *Mécanique Analytique*. Courcier, 1811.

[Lam13] Peter Lammich. Automatic Data Refinement. *Archive of Formal Proofs*, 2013, 2013.

[Lan09] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. B.G. Teubner, 1909.

[LaS60] Joseph LaSalle. Some Extensions of Liapunov's Second Method. *IRE Transactions on Circuit Theory*, 7(4):520–527, Dec 1960.

[LaS76] Joseph LaSalle. *The Stability of Dynamical Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1976.

[LaV06] Steven M. LaValle. *Planning algorithms*. Cambridge University Press, 2006.

[LDF+18] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.07 - Documentation and user's manual*, 2018.

[Leb50] Henri Lebesgue. *Leçons sur l'Intégration et la Recherche des Fonctions Primitives*. Gauthier-Villars, Imprimeur-Éditeur, second edition, 1950.

[Lel15] Catherine Lelay. *Repenser la bibliothèque réelle de Coq : vers une formalisation de l'analyse classique mieux adaptée. (Reinventing Coq's Reals library : toward a more suitable formalization of classical analysis)*. PhD thesis, University of Paris-Sud, Orsay, France, 2015.

[Lem06] Lemma 1 Ltd. *ProofPower HOL Reference Manual*, 2006.

[Les07] David R Lester. Topology in PVS: Continuous Mathematics with Applications. In *Proceedings of the Second Workshop on Automated Formal Methods*, AFM '07, pages 11–20, New York, NY, USA, 2007. ACM.

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)*. PhD thesis, University of Paris-Sud, Orsay, France, 2004.

[Let08] Pierre Letouzey. Extraction in Coq: An Overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[LFB00] Rogelio Lozano, Isabelle Fantoni, and Dan Block. Stabilization of the inverted pendulum around its homoclinic orbit. *Systems & Control Letters*, 40(3):197–204, 2000.

[Lia07] Alexandre Liapounoff. Problème général de la stabilité du mouvement. *Annales de la Faculté des sciences de Toulouse : Mathématiques*, 9:203–474, 1907.

[LL19] Peter Lammich and Andreas Lochbihler. Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches. *Journal of Automated Reasoning*, 63(1):53–94, 2019.

[LM12] Catherine Lelay and Guillaume Melquiond. Différentiabilité et intégrabilité en Coq. Application à la formule de d'Alembert. In *23èmes Journées Francophones des Langages Applicatifs*, pages 119–133, Carnac, France, 2012.

[LMCLD] The Lean Mathematical Components Library Developers. The Lean Mathematical Components Library. https://github.com/leanprover/mathlib.

[Mag18] Marco Maggesi. A Formalization of Metric Spaces in HOL Light. *Journal of Automated Reasoning*, 60(2):237–254, 2018.

[Map19] Maplesoft, a division of Waterloo Maple Inc. Maple 2019. https://www.maplesoft.com/products/Maple/, 2019.

[Mat62] Vladimir Matrosov. On the Stability of Motion. *Journal of Applied Mathematics and Mechanics*, 26(5):1337 – 1353, 1962.

[May01] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, décembre 2001.

[May02] Micaela Mayero. Using Theorem Proving for Numerical Analysis (Correctness Proof of an Automatic Differentiation Algorithm). In Victor Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2002.

[May12] Micaela Mayero. *Problèmes critiques et preuves formelles*. Habilitation à diriger des recherches, Université Paris 13, novembre 2012.

[MBG06] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51, 2006.

[MC08] Sayan Mitra and K. Mani Chandy. A Formalized Theory for Verifying Stability and Convergence of Automata in PVS. In Mohamed et al. [MMT08], pages 230–245.

[MCT] The Mathematical Components Team. The Mathematical Components Library. https://github.com/math-comp/math-comp/releases/tag/mathcomp-1.9.0. Version 1.9.0.

[MG06] José Luis Mancilla-Aguilar and Rafael Antonio García. An extension of LaSalle's invariance principle for switched systems. *Systems & Control Letters*, 55(5):376–384, 2006.

[Mil85] Robin Milner. The Use of Machines to Assist in Rigorous Proof. In *Proceedings Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 77–88, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[MLS94] Richard M. Murray, Zexiang Li, and Shankar Sastry. *A Mathematical Introduction to Robotics Manipulation*. CRC Press, 1994.

[MM.37] MM. les secrétaires perpétuels. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, volume 205. Gauthier-Villars, Imprimeur libraire, 1937.

[MMT08] Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.

[Mon76] J. Donald Monk. *Mathematical Logic*. Graduate Texts in Mathematics. Springer-Verlag, 1976.

[MP11] Jean-Marie Madiot and Pierre-Marie Pédrot. Constructive axiomatic for the real numbers. In Bas Spitters, editor, *Coq Workshop 2011, Nijmegen, The Netherlands, August 26, 2011*, Aug 2011. Extended abstract.

[MPW+18] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: minimizing the Coq extraction TCB. In Andronick and Felty [AF18], pages 172–185.

[MR91] QingMing Ma and John C. Reynolds. Types, Abstractions, and Parametric Polymorphism, Part 2. In Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA, March 25-28, 1991, Proceedings*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 1991.

[MR17] Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Bertot and Vafeiadis [BV17], pages 90–99.

[MRAL16] Gregory Malecha, Daniel Ricketts, Mario M. Alvarez, and Sorin Lerner. Towards Foundational Verification of Cyber-Physical Systems. In *2016 Science of Security for Cyber-Physical Systems Workshop, SOSCYPS@CPSWeek 2016, Vienna, Austria, April 11, 2016*, pages 1–5. IEEE Computer Society, 2016.

[MS13] Evgeny Makarov and Bas Spitters. The Picard Algorithm for Ordinary Differential Equations in Coq. In Blazy et al. [BPP13], pages 463–468.

[MT13] Assia Mahboubi and Enrico Tassi. Canonical Structures for the Working Coq User. In Blazy et al. [BPP13], pages 19–34.

[MT18] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at https://math-comp.github.io/mcb/book.pdf, 2018. With contributions by Yves Bertot and Georges Gonthier. Version of 2018/08/11.

[New87] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. Londini, Jussu Societatis Regiæ ac Typis Josephi Streater. Prostat apud plures Bibliopolas., 1687.

[NK09] Adam Naumowicz and Artur Korniłowicz. A Brief Overview of Mizar. In Berghofer et al. [BNUW09], pages 67–72.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[Paş08] Ioana Paşca. A Formal Verification for Kantorovitch's Theorem. In *Journées Francophones des Langages Applicatifs*, January 2008.

[Paş11] Ioana Paşca. Formal proofs for theoretical properties of Newton's method. *Mathematical Structures in Computer Science*, 21(4):683–714, 2011.

[PD90] Beata Padlewska and Agata Darmochwał. Topological Spaces and Continuous Functions. *Formalized Mathematics*, 1(**1**):223–230, 1990.

[Pet82] Linda Petzold. Differential/Algebraic Equations are not ODE's. *SIAM Journal on Scientific and Statistical Computing*, 3(3):367–384, 1982.

[Pot08] Loïc Pottier. Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[PQ08] André Platzer and Jan-David Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008.

[PT98] Loïc Pottier and Laurent Théry. gbcoq: Certified Gröbner bases computations. http://www-sop.inria.fr/croap/CFC/Gbcoq.html, 1998.

[Rey83] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Rey84] John C. Reynolds. Polymorphism is not Set-Theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[Rou18] Damien Rouhling. A Formal Proof in Coq of a Control Function for the Inverted Pendulum. In Andronick and Felty [AF18], pages 28–41.

[RTC11] RTCA Inc. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, Dec 2011.

[Sai99] Amokrane Saibi. *Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory.* Theses, Université Pierre et Marie Curie - Paris VI, March 1999.

[Scha] Daniel Schepler. coq-topology: Topology Library for Coq. https://github.com/coq-contribs/topology.

[Schb] Daniel Schepler. coq-zorns-lemma: Set Theory Library for Coq. https://github.com/coq-community/zorns-lemma.

[SD19] The Sage Developers. *SageMath, the Sage Mathematics Software System*, 2019. Version 8.7.

[Sim04] Carlos Simpson. Computer Theorem Proving in Mathematics. *Letters in Mathematical Physics*, 69(1):287–315, Jul 2004.

[Sko01] Bartłomiej Skorulski. The Tichonov Theorem. *Formalized Mathematics*, 9(**2**):373–376, 2001.

[SMKT96] Naoji Shiroma, Osamu Matsumoto, Shuuji Kajita, and Kazuo Tani. Cooperative Behavior of a Wheeled Inverted Pendulum for Object Transportation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS 1996, November 4-8, 1996, Osaka, Japan*, pages 396–401. IEEE, 1996.

[SN08] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Mohamed et al. [MMT08], pages 28–32.

[SNI02] Tomomichi Sugihara, Yoshihiko Nakamura, and Hirochika Inoue. Realtime Humanoid Motion Generation through ZMP Manipulation Based on Inverted Pendulum Control. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation, ICRA 2002, May 11-15, 2002, Washington, DC, USA*, pages 1404–1409. IEEE, 2002.

[SO99] Natarajan Shankar and Sam Owre. Principles and Pragmatics of Subtyping in PVS. In Didier Bert, Christine Choppy, and Peter D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT '99, Château de Bonas, France, September 15-18, 1999, Selected Papers*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 1999.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Mohamed et al. [MMT08], pages 278–293.

[SPA16] Giordano Scarciotti, Laurent Praly, and Alessandro Astolfi. Invariance-Like Theorems and "lim inf" Convergence Properties. *IEEE Transactions on Automatic Control*, 61(3):648–661, March 2016.

[ST18] Matthieu Sozeau and Nicolas Tabareau, editors. *Coq Workshop 2018, Oxford, UK, July 8, 2018*, 2018.

[Str] Pierre-Yves Strub. multinomials: Multinomials for Ssreflect. https://github.com/math-comp/multinomials.

[SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[TCT] The COQTAIL Team. The COQTAIL Library. https://github.com/coqtail/coqtail.

[Thé] Laurent Théry. grobner: A formalisation of Gröbner basis in ssreflect. https://github.com/thery/grobner.

[Thé01] Laurent Théry. A Machine-Checked Implementation of Buchberger's Algorithm. *Journal of Automated Reasoning*, 26(2):107–137, 2001.

[TTS18] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proceedings of the ACM on Programming Languages*, 2(ICFP):92:1–92:29, 2018.

[UFP13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Wad89] Philip Wadler. Theorems for Free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.

[Wei16] Ittay Weiss. The Reals as Rational Cauchy Filters. *New Zealand Journal of Mathematics*, 46:21–51, 2016.

[Wie06] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[Wie12] Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85–100, 2012.

[Wil08] Albert Wilansky. *Topology for Analysis*. Dover books on mathematics. Dover Publications, New York, NY, 2008.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.