ÉCOLE POLYTECHNIQUE

École Doctorale de l'École Polytechnique

Laboratoire d'Informatique de l'École Polytechnique

Inria Saclay – Île-de-France

Inria – Microsoft Research

THÈSE DE DOCTORAT

par **Cyril Cohen**

soutenue le **20 novembre 2012**

Pour obtenir le grade de **Docteur de l'École Polytechnique**

Discipline : **Informatique**

Formalized algebraic numbers: construction and first-order theory

Président du Jury

M. Barthe Gilles	*Researcher, IMDEA*

Directeurs de thèse

Mme Mahboubi Assia	*Chargée de Recherche, Inria*
M. Werner Benjamin	*Directeur de Recherche, Inria*

Rapporteurs

M. Avigad Jeremy	*Professor, Carnegie Mellon University*
M. Filliâtre Jean-Christophe	*Chargé de Recherche, CNRS*
Mme Roy Marie Françoise	*Professeur, Université Rennes 1*

Examinateurs

M. Gonthier Georges	*Principal Researcher, Microsoft Research*
M. Spitters Bas	*Researcher, University of Nijmegen*

À Cécilia et à mes parents,

Je ne pourrais pas commencer cette thèse sans remercier infiniment ma directrice de thèse, Assia, de m'avoir encadré pendant plus de trois ans. Merci pour ton inconditionnel soutien, tes conseils, tes nombreuses relectures, ta gentillesse et ta patience.

Je remercie aussi Georges de m'avoir si souvent guidé dans mon travail et donné des retours éclairés sur mes travaux. Merci aussi pour ta persévérance à expliquer et ré-expliquer gentiment toutes ces choses qui te semblent si triviale.

Je souhaite remercier Benjamin d'avoir avoir pris la responsabilité de m'encadrer officiellement, mais aussi pour les TDs de SSReflect que j'ai eu l'occasion d'enseigner à tes côtés à l'X.

Je remercie Jean-Christophe Filliâtre et Marie-Françoise Roy d'avoir accepté de rapporter ma thèse, et je vous remercie aussi pour vos retours sur ma thèse, qui m'ont aidé à l'améliorer.

Thank you Jeremy Avigad for accepting to review my thesis, and for your comments that helped me improving it.

Je remercie Gilles Barthe d'avoir accepté de faire partie de mon jury.

Thank you Bas Spitters for having accepted to be in my jury, and also for the nice stay in Nijmegen.

Merci Thierry Coquand, pour ton accueil à Göteborg et les intéressantes discussions que nous avons eu.

Je remercie tous mes collègues et camarades de Mathematical Components, de Typical et de Marelle : Enrico, Bruno, Stéphane, Arnaud et Chantal ainsi que Laurence, Laurent, Yves, Loïc, Ioana et Guillaume pour la bonne ambiance que vous avez su maintenir et alimenter à tout instant. Je souhaite remercier plus particulièrement mes nombreux co-bureaux de quelques semaines ou de plusieurs années avec qui j'ai eu l'occasion de troller ou discuter au quotidien et qui ont dû supporter mes quarts d'heures de folie : François, Denis, Mathieu, Bruno, Vincent, Maxime, Russell, Thomas et Victor.

Je remercie également les collègues du labo commun Inria-MSR pour avoir maintenu une vie de laboratoire chaleureuse et des discussions de café et de repas animées : Jean-Jacques, Frédéric, Bruno, Alin, Damien, Marc, Alexandre, Elie, Pierre-Yves, Iro et Jeremy. Une mention spéciale à Pierre : dans tes tentatives sans cesse renouvelées à vouloir trivialiser mon travail de thèse, tu m'as déniché un papier dont je rêvais [34].

Merci aux collègues de Proval (Christine, Guillaume, Sylvie et Catherine) pour votre accueil au LRI puis au PCRI, et de πr^2 (Matthieu, Hugo, Pierre, Pierre, Stéphane, Mathias et Alexis) pour votre accueil à Place d'Italie.

Thank you Swedish people for your repeted welcome in Chalmers : Anders, Guilhem, Simon, Bassel, Ana, Peter and Jonna.

Merci aux collègues du DIX avec qui j'ai eu le plaisir d'enseigner : Olivier, Dominique, Kartik, David, Jonathan, Christophe et Baptiste.

Merci beaucoup à Lydie puis à Valérie ainsi qu'à Martine pour avoir toujours su m'accueillir avec le sourire, malgré tout ce que j'ai eu à vous demander, et pour avoir réglé tous mes problèmes avec une efficacité remarquable.

Je n'en serais pas là sans ceux dont les enseignements ont marqué ma vie.

Merci à Nicolas Tosel, pour m'avoir initié à l'algèbre.

Merci à Hubert Comon, pour m'avoir aidé à faire les bons choix d'orientation et aussi pour m'avoir initié à la logique.

Merci à Jean Goubault-Larrecq, pour m'avoir initié à la sémantique.

Merci à Paul-André Méliès, pour m'avoir orienté vers un sujet de stage de M2 et de thèse qui m'a profondément inspiré.

Durant cette thèse, j'aurais pu me perdre si je n'avais pas eu l'appui d'amis formidables, que je souhaite remercier. Je pense notamment à Ouardane, avec qui j'ai réalisé ou échoué tant de projets, mais toujours dans la bonne humeur. Je pense aussi à mes amis Cachanais, syndiqués, informaticiens, magiciens et improvisateurs avec qui j'ai partagé tellement de choses ces dernières années : merci (dans un ordre plus ou moins arbitraire) Claire, Florence, Drébon, S., Ayman, Ryo, Romain, Sasso, François, Marianne, Michel, Picpic, Zerty, FloFlo, Anne, Carlos, Marie, Seb, Pauline, Minipouce, Théa, Antony, Sam, Fanny, Nico, Justine, Pat, Gaëtan, Moubi, CPA, S', Antony, Barry, Olivier, Eudes, Mathilde et Aline.

Merci aussi à tous les gens qui mériteraient d'être sur cette page mais que j'ai honteusement oubliés. Je vous présente mes excuses les plus sincères.

Merci, cher lecteur, ça fait plaisir d'être lu.

Dear reader, thank you for reading this thesis.

Merci à ma famille, notamment à mes parents, vous ne m'avez pas beaucoup vu ces dernières années, mais c'est grâce à vous que j'en suis là et je ne l'oublie pas.

Last but not least, merci Cécilia, pour avoir supporté non sans douleur ma vie de doctorant désorganisé, et tant d'autres choses.

Contents

I Infrastructure 9

1 Booleans in Coq logic 11
- 1.1 Reflection . . . 11
- 1.2 Excluded middle and proof irrelevance . . . 14
- 1.3 Interaction between bool and other types . . . 15

2 Algebraic Hierarchy 17
- 2.1 On the meaning of *axiom* . . . 17
- 2.2 Choice of interfaces . . . 18
- 2.3 Structure inference . . . 20

3 Polynomials 25
- 3.1 Polynomial arithmetic . . . 25
- 3.2 Resultant . . . 27

II Construction of numbers 29

4 Numeric rings 31
- 4.1 Extending the hierarchy . . . 31
  - 4.1.1 The Numeric Hierarchy . . . 31
  - 4.1.2 Discussion on the interfaces and their names . . . 34
- 4.2 Signs, case analysis based on comparisons for reals . . . 35
- 4.3 Intervals . . . 40
- 4.4 Structure of integers and rational numbers . . . 44
  - 4.4.1 Integers . . . 44
  - 4.4.2 Rational numbers . . . 44

5 Cauchy reals, algebraics **47**
5.1 Cauchy reals . . . 48
5.1.1 Cauchy sequences . . . 48
5.1.2 Equality on Cauchy reals . . . 49
5.1.3 Arithmetic operations and bounding . . . 51
5.1.4 Reasoning with big enough values . . . 53
5.1.5 Comparison with other implementations of reals . . . 56
5.2 Algebraic reals . . . 56
5.2.1 Decidability of comparison . . . 56
5.2.2 Arithmetic operations . . . 57

6 Quotient types in Coq **61**
6.1 Quotients in mathematics . . . 61
6.2 An interface for quotients . . . 62
6.3 Quotient by an equivalence relation . . . 68
6.3.1 Quotient of a choice structure . . . 69
6.3.2 Quotient of type with an explicit encoding to a choice structure 71
6.4 Related work on quotient types . . . 72

7 Construction of the real closure as a type **75**
7.1 Algebraic numbers have an explicit encoding to a choice type . . . 75
7.1.1 Decoding to algebraic Cauchy reals . . . 76
7.1.2 Encoding of algebraic Cauchy reals . . . 77
7.2 A quotient type for algebraic numbers . . . 77
7.2.1 Construction of the quotient type . . . 77
7.2.2 Real algebraic numbers form a real closed field . . . 79

8 Discussion on the algebraic closure **81**
8.1 Equivalent definitions for real closed fields . . . 81
8.2 Direct construction and other methods . . . 89

III Theory of real and algebraically closed fields **91**

9 Elementary polynomial analysis **93**
9.1 Direct consequences of the intermediate value theorem . . . 93
9.2 Root isolation . . . 95
9.3 Root neighborhoods . . . 96
9.4 About existential formulas . . . 99

10 Syntax, semantics and decidability **101**
10.1 First-order logic, the usual presentation . . . 101
10.2 Formalizing first-order logic in Coq . . . 103
10.3 Quantifier elimination by projection . . . 107
10.4 Decidability . . . 109

11 Solving polynomial systems of equations (in one variable) **111**
11.1 Reduction of the system . . . 112
11.1.1 For discrete algebraically closed fields . . . 112

11.1.2 For discrete real closed fields . . . 114
11.2 Root counting using Tarski queries . . . 115
11.3 Cauchy index . . . 119
11.4 Algebraic formula characterizing the existence of a root . . . 123

12 Programming and certifying the quantifier elimination 125
12.1 An example . . . 125
12.2 Algorithm transformation and projection . . . 128
12.3 Direct transformation . . . 129
12.4 Continuation passing style transformation . . . 131

IV Conclusion and perspectives 135

Lists of Figures 143

Bibliography 145

Index 151

Introduction

The formalization of mathematics is the action of explaining definitions, theorems and proofs of those theorems to a software, called a *proof assistant*. A proof assistant behave like a calculator: the user writes an expression (a formula, a definition, ...) in the language of the assistant and it computes a result, which tells whether the expression is valid. In order to understand the level of confidence we can expect from this system, it is worth mentioning there is a small piece of code, called the *kernel* which sole task is to check a fully detailed proof. If this short piece of code is bug-free (and if the compiler that compiles it, is bug-free too) then any proof that passes its checks is guaranteed to be correct. We call this piece of code the trusted base, because it is the only part one needs to trust in order to trust everything else. It is essential that the trusted base should be readable by a human being and should be kept as small as possible, in order to avoid the probability of error. In particular its size must be negligible in front of the size of all the formalizations that are made on top of it. On top of the kernel many tools are built which do not need to be bug-free since anything that has to be check is sent to the kernel in the end. These tools may include a proof mode which provides *tactics* to help the user build a proof. For assistants based on type theory, there would be a type inference mechanism, which task is to reconstruct the types of the object the user gives, in order to let him be more concise. There could also be a notation system to help the user provide a mathematical look and feel to its formalization. A proof assistant requires a formal proof for any simple fact, even those for which a human would believe without questioning (sometimes wrongly).

The idea of computing on proofs in the same way as on numbers is not new: indeed, Gottfried Wilhelm Leibniz (1646 — 1716) imagined the "calculus ratiocinator", a machine that could mechanize logic. However, we had to wait until Friedrich Ludwig Gottlob Frege's 1879 *Begriffsschrift* for logic to be written in a way that could actually be mechanized. Automath, devised by Nicolaas Govert de Bruijn in 1967, was the first logical framework to be implemented in practice. Since then, the world of logical framework and proof assistants flourished [88], and several major results in standard mathematics were formalized, among them we can cite the Prime Number Theorem [3] in Isabelle/HOL, the Fundamental Theorem of Algebra [40] and the Fundamental Theorem of Calculus [32] in Coq, and the Four Colour theorem [41]

in Coq/SSReflect.

The proof assistant we use in this thesis work is Coq [85, 84]. In Coq, there is a proof mode: the user can visualize the current state of the proof, *i.e.* what remains to be proved, and progress using *tactics*. A state in this proof mode is called a *goal*, which is composed of a *context* – a set of known facts, such as the hypotheses of the fact we are proving – and a *conclusion* which is what we have to show using the context. Each tactic can either make the current goal progress, by adding new data or new facts to the context or changing the conclusion, or it can split the current goal in several subgoals. Coq logic is based on an extension of the Calculus of Constructions (CoC) [29] (designed in 1986): the Calculus of Inductive Constructions (CIC) [72] with Universes. Unlike the classical logic that most mathematicians use, Coq logic is constructive, which means that the law of excluded middle is not valid. An interpretation is that not all propositions are either provable or have a provable negation (as Gödel first incompleteness theorem shows). Another consequence of constructivity is that proving an existential statement $\exists x,\ P\ x$ amounts to providing an explicit witness x_0 such that $P\ x_0$ holds. For a statement of the form $\forall y,\ \exists x,\ P\ x\ y$, it amounts to writing a program that given y as an argument, outputs an x such that $P\ x\ y$ holds. This illustrates a piece of the correspondence between proofs and programs, known as Curry-Howard correspondence. This correspondence is a key feature of Coq logic, making the language for writing expressions in Coq both suitable for expressing logic formulas and for programming. As a programming language, Coq is a typed language, with a strong type system where types can depend on arbitrary expressions. Those types are called *dependent types*, and it makes possible to define the type of matrices of size n.

Whereas such a constructive system can be seen as restrictive, it is in fact more expressive than a classical system. Indeed, any proposition that can be proved classically can at least be proved in Coq under classical axioms. Moreover, providing a constructive proof for a theorem is even stronger than providing a classical one, since it requires to write the program that solves a problem, and it can provide more insight to the author and to the reader. Additionally, a programming language brings *computation* to the system, and a strong type system brings *type inference*. Indeed, typed languages may rely on type inference in order to help the user to be concise. Without type inference, a user would have to write systematically the type of any object he uses, which we absolutely want to avoid as mathematicians. Indeed, mathematical objects often come with a huge quantity of implicit information which would become unreadable if made explicit. For example, when we say "let R be a field" in the language of mathematics, we implicitly consider the field operators and the properties of field, but we do not write all the details. With computation, one can write a Coq program that calculates on arithmetic or symbolic expression, just like any other programming language. It can even compute on fragments of its own logic, by a process we call *reflection*. The notion of reflection could already be found in the work of Pfenning and Paulin-Mohring [72] as a deep embedding of a calculus (F_2 to be precise) inside Coq, but its applications as a tool to build formal proof producing decision procedures for mathematical theories had to wait until a work by Boutin [17]; we call this methodology *large scale reflection*. In this thesis, we use intensively the *small scale reflection* [44, 46]. It was developed for the proof of the Four Colour theorem [41] and further use of this methodology led to the creation and the development of the SSReflect extension for Coq, leaving

its core and logic intact, but extending the tactic language and providing a library of mathematical results formulated in a reusable way. More recently, SSReflect was used to formalize a complete proof of Feit-Thompson Theorem [42]; this thesis contributed to this achievement.

Any formalization about mathematics in Coq has to take all of these into account: the drawbacks and advantages of constructiveness and of a strong type system. Hence, as of today, formalizing mathematics is a work that requires the knowledge of the type theory which constitutes the logic, skills in programming and to be at ease with the mathematics that are at stake. In my experience, a difficulty with this last point is that paper proofs often have gaps in them, like something non trivial which is not even mentioned, or even something which is locally wrong. Another frequent problem is to find constructive alternatives, which are often not widespread. A good knowledge of the type theory is mandatory to know which formulas are typable and which are not, which is the first stage of the coding of a theory: even before trying to prove something, its statement has to be accepted by the system as well-formed. Programming, and more specifically the organization of the code, the refactoring of the code, the design of correct algorithms, the completeness and the documentation of libraries is the biggest task of all in my point of view: it is the most difficult, important and time consuming. Not only does it require to have both a good understanding of the type theory and of the underlying (constructive) mathematics of what is formalized, but also to have a good idea of how people — apart from I and my coworkers — may want to use what is formalized.

John Harrison [51] made the following observation: "There isn't a perfect match between what humans find easy and what proof assistants find easy, so it's possible that a proof that a human would regard as more difficult is actually easier to formalize, even setting aside a comparison of the amount of knowledge assumed." Surprisingly, very simple and old mathematical results can require more effort than recent results, just because they are used more extensively and in so many different context. The part of my work I spent the most time on would be the most simple thing from a mathematician's point of view. Indeed, I developed, reorganized, refactorized the development of the numeric hierarchy (see Chapter 4) for at least a year, and I revised the development about quotient types at least five times over three years (see Chapter 6) while some developments about more complicated mathematics only took me a few days, weeks or months (e.g. a few weeks for the construction of real algebraic numbers). The mathematical results we formalized are not very advanced, they can be partially or fully taught to undergraduate students, up to the constructiveness of the proofs. Hence we provide two little contributions to constructive mathematics which are two important contributions to the formalization of mathematics, not only for their final result, but also for the methodologies and tools that we created to reach those.

On the mathematical content of this thesis

The mathematics used in this thesis are mainly about discrete real and algebraically closed fields, from the construction of such fields to the study of one of their most fundamental property: the decidability of their first-order theory [83]. Indeed, the two main contributions of this thesis are the full constructive formalization in the Coq proof assistant of the real closure of a discrete Archimedean field, and the

certification of a quantifier elimination procedure for the theories of algebraically and real closed fields.

The real closure of the discrete Archimedean field of rational numbers is the field of real algebraic numbers, and so my first contribution is essentially about algebraic numbers and the study of their properties. Algebraic numbers (respectively real algebraic numbers) are part of the mathematical folklore, and there is not much to say about their classical construction: they are the complex (respectively real) numbers that are roots of a polynomial with rational coefficients. However, their intuitionistic construction is more intricate and, up to our knowledge, was never formalized in a constructive proof assistant. Algebraic numbers also have very interesting properties: they form a countable subset of complex numbers and can be substituted for usual complex numbers in developments that require only their first-order properties, such as in the formalized proof of the Feit-Thompson Theorem for example. Finally, algebraic numbers also form a discrete algebraically closed field, and real algebraic numbers a discrete real closed field, which makes the second contribution a study of the theory of algebraic numbers and real algebraic numbers. Part II (and more precisely Chapter 5 and 7) is devoted to the construction of algebraic numbers and to the proof they form a real closed field.

The construction we detail and their properties are described in many standard references on constructive mathematics [63] or in computer algebra [15]. However, the constructive proofs we used are difficult to find in the literature because classical mathematics books tend to provide classical proofs, and even constructive mathematics books sometimes rely on classical arguments. Thus, the implementation of these results in a proof assistant requires various changes in their presentation to make them constructive. Hence, our development is not a literate translation of a well-chosen reference, but is rather a synthesis of poorly known results from the mathematical folklore. Then, we study the relation between real algebraic numbers and their algebraic closure and the construction of complex algebraic numbers from real algebraic numbers, in Chapter 8. The latter construction involves a generalized version of the fundamental theorem of algebra, for which we give several algebraic constructive proofs, and formalized one.

The second contribution, formalizing quantifier elimination, is seen as a consequence from basic results in real algebraic geometry. Real algebraic geometry is the study of sets of roots of multivariate polynomials in real closed fields. We formalize a proof of quantifier elimination for the first-order theory of real closed fields, which in turn entails the decidability of the first-order theory of real closed fields. Since the original work of Tarski [83] who first established this decidability result, many versions of a quantifier elimination algorithm have been described in the literature. Shortly after Tarski published his proof, Seidenberg published another proof [78]. An elementary, algebraic proof is the one described by Hörmander following an idea of Paul Cohen [27, 56, 14]. However, the first algorithm with a "reasonable" upper bound (doubly exponential in the number of quantifiers) for deciding the theory of real closed field was obtained by Collins [28], using Collins's Cylindrical Algebraic Decomposition algorithm. At the time we write this thesis, it is the *implemented* algorithm which has the best complexity. However, Gregor'ev and Vrobjov introduced a breakthrough by solving the existential problem of decision for the theory of real closed fields, with a complexity of a simple exponential in the number of quantifiers [50]. This led Grigor'ev to solve the general decision

problem in doubly exponential complexity [49], where the second exponent is the number of alternations of quantifiers. Heintz, Roy and Solernó [53] and Renegar [75] improved independently the latter result.

All the algebraic proofs rely on the same central idea of the existence of a projection: starting from a family $\mathcal{P} \subset \mathbb{R}[X_1, \ldots, X_{n+1}]$, compute a new family $\mathcal{Q} \subset \mathbb{R}[X_1, \ldots, X_n]$, where the variable X_{n+1} has been eliminated. The recursive use of this projection leads to a quantifier elimination procedure. The computational efficiency of this projection dominates and hence governs the complexity of the quantifier elimination. In the case of the algorithm of Cohen-Hörmander, the projection is rather naive and only uses repeated Euclidean divisions and simple derivatives. The breakthrough introduced by the Collins' algorithm is due to a clever use of better a better way to compute Cauchy indexes (defined in Section 11.3), namely subresultant polynomials, and of partial derivatives in order to improve the complexity of the projection. The quantifier elimination algorithm we describe in Part III (and more precisely in Chapters 11 and 12) has a naive complexity (a tower of exponentials in the number of quantifiers).

We follow the presentation given in the second chapter of [5], based on the original method of Tarski [83]. This algorithm is more intricate than the one of Hörmander, and closer to Collins' one with respect to the objects it involves. Objects like signs at a neighborhood of a root, pseudo-remainder sequences or Cauchy bounds are indeed crucial to both algorithms and hence part of the present formalization. Moreover, we give a simple quantifier elimination procedure for algebraically closed field, the explanation of which is given in Section 11.1.1, also following a presentation given in [5]. Thanks to this, first-order formulas on real algebraic numbers and complex algebraic numbers are proven decidable in Coq.

The libraries, tools and methodologies we used

Those two mathematical contributions both relied on and involved the development and achievement of several tools, abstractions, and theories. The most important of these are:

- The extension of the theory of divisibility on polynomials, as we briefly explain in Section 3.1.

- The construction of a new hierarchy of discrete numeric algebraic structures, including ordered, Archimedean and real closed fields. This new hierarchy comes with its theory and facilities to use it. This library, presented in Chapter 4, is currently used in the character theoretic part of the proof of the Feit-Thompson Theorem. We also instantiated these structures with integers and rational numbers.

- The formalization of a weak form of quotients (see Chapter 6), which helped us construct algebraic numbers. It was also used by other contributors to the SSReflect library, in order to formalize Galois Theory or the construction of the algebraic closure of countable field.[1]

[1] both of which are not published yet

- The development of a methodology to deal with "big enough numbers" in analytic reasoning and the construction of a small and simple representation of constructive real numbers using Cauchy sequences, described in Chapter 5.

- The development of the elementary theory of real closed fields, which is basically a limited form of polynomial analysis, treated in a constructive way, in Chapter 9.

- A style for programming and certifying algorithm written in continuation passing style, in Chapter 12.4.

On Coq and SSReflect

Coq is a proof assistant based on the Calculus of Inductive Constructions with Universes. The *Vernacular* language gives a set of commands to organize the contents of a file, like declaring a definition, a lemma, an inductive type and various hints for tools that are not in the kernel. The *Gallina* language is the language of Coq expressions. It is used to write programs and their types. Following the Curry-Howard correspondence, types are statements of theorems and their proofs are programs whose type is the statement. Writing a proof is in fact writing a program. We call these programs *proof terms*. The tactic language is used to build proofs without writing proof terms by hand.

The SSReflect extension of Coq brings both an alternative set of tactics and a library. The two of them are quite independent, but the library is designed to be used with the SSReflect tactics. The library is not only constituted of definitions, lemmas and theorems but also uses advanced features of the Coq system, such as notations, implicit arguments, hints, coercions and canonical structures. The SSReflect tactic language and library were initially designed to prove the Four Colour theorem [41]. But since then, the language and the library have been improved in order to tackle the Feit-Thompson Theorem (also known as Odd Order Theorem) [42]. The library has been extended with group theory [45], but also algebraic structures [36], polynomials and matrices [43, 11, 67], representation theory [12] and character theory, which are the prerequisites to formalize the two books [6, 70] that regroup and summarize the proof of the Feit-Thompson Theorem. One of the choices made in the SSReflect library is never to rely on Coq axioms, which are statements that are admitted. Indeed, it is not uncommon to add the excluded middle to Coq like this, and in a past version of Coq this could lead to inconsistency [20, 73]. We stay faithful to this choice of not having Coq axioms and we never rely on them.

In this thesis, we do not provide an introduction to Coq. For that purpose, we recommend the reader to look at the appropriate references. For an introduction to Coq, he may read [7]. For a description of the Coq features that are mainly used in SSReflect, we recommend the reading of [37]. Instead, we will only introduce the SSReflect tools, methodologies, tactics and definitions which are needed to understand this work.

Organization of the thesis

The thesis is organized in three parts. I made only minor contributions to the formalizations described in the first part (only small corrections, documentation and a few results about polynomial division and bivariate polynomials). The second part and the third part are entirely original contributions, except for Chapter 10.

Part I introduces the material on top of which this thesis was build.

Chapter 1 explains the central use of Booleans in the SSReflect library and hence in our work. Although it is one of the simplest datatypes one can find, their interpretation as truth values leads to a completely different way of carrying out proofs in Coq.

Chapter 2 and Chapter 3 describe respectively the SSReflect algebraic hierarchy and the polynomial library, which remained essentially the same since the beginning of this thesis' work. For the sake of readability, we only detail what is necessary to understand this thesis. A more complete description of those can be found by reading the generated coqdoc of the SSReflect libraries.

Part II describes the construction of algebraic numbers and all the tools and definitions that we had to formalize in order to ease this construction and to integrate it smoothly in the SSReflect library. Earlier versions of the work presented in this part have been published [23, 22, 21, 24].

Chapter 4 introduces an extension of the algebraic hierarchy with order relation and norm operators, which must be compatible with the ring or field operations: the numeric hierarchy. We details some of the tools we designed to reuse the theory we developed about this hierarchy. We also provide the integers and rational numbers as instantiation of the theory of numeric domains and numeric fields.

Chapter 5 both describes a small and self-contained formalization of real numbers using Cauchy sequences and the subtype of reals that are algebraic. In particular, we developed a tool for reasoning with "big enough numbers" that eased a lot the proofs in these theories.

Chapter 6 explains the methodology we follow when dealing with quotient types, and the construction of a quotient using an equivalence relation or an ideal.

Chapter 7 combines all the results from the two previous chapters to build the type of real algebraic numbers. This type is then used to formalize the complex algebraic numbers in Chapter 8, as needed in the formalization of Feit-Thompson Theorem.

Part III studies the theory of real and algebraically closed fields, it only depends on the previous part through Chapter 4. Indeed, the theory of those structure is independent from their implantation. The main intention for this part is to describe the quantifier elimination procedure on real closed fields and explain the formalization of its proof. Earlier versions of the work presented in this part have been published [25, 26].

Chapter 9 explores the consequences of the axioms of real closed fields, by building a root isolation procedure and studying the notion of neighborhood of a number.

Chapter 10 describes the objects on which the quantifier elimination procedure operates, thereby introducing basic notions about first-order logic and model theory.

Chapter 11 details a way to solve a system of equations and inequations. Statements and proofs are inspired from our main reference on the subject [5].

Finally, Chapter 12 combines all the results from the previous chapters into the programming and certification of a quantifier elimination procedure for real closed fields.

The sources of the formalization

Some of the sources of the formalization presented in this thesis have been integrated to the 1.4 release of SSReflect (which is the latest release at the time we wrote this thesis). The rest of the source code is in the private repository of the Mathematical Components team [74], but I made my files available on my personal web page, at the following url: http://perso.crans.org/cohen/phd/.

At the time we wrote this, the source code related to this thesis contained more than 15000 lines of code, including 584 Definitions and 1951 Lemmas, Facts and Theorems.

Let us describe the correspondence between the chapters and the source code files.

Chapter 4 describes the files `ssrnum.v`, `ssrint.v` and `rat.v` of the 1.4 release of SSReflect, and the interval theory (Section 4.3), which is in the non released file `interval.v`.

Chapter 5 and Chapter 7 describe non released code, which can be found in `cauchyreals.v` for the construction of Cauchy reals (Section 5.1), `bigenough.v` for reasoning with big enough numbers (Section 5.1.4), and `realalg.v` for the rest.

Chapter 6 corresponds to the file `generic_quotient.v` in the 1.4 release of SSReflect.

Chapter 8 is mostly describing proofs that have not been formalized. The file `complex.v` contains a construction of the complex algebraic numbers, using a proof of the Fundamental Theorem of Algebra relying on linear algebra (Section 8.2).

Chapter 9 describes the unreleased files `closed_field.v` (for quantifier elimination on algebraically closed fields), `polyorder.v` and `polyrcf.v` (for polynomial analysis on real closed fields), Chapter 11 the unreleased file `qe_rcf_th.v` and Chapter 12 the unreleased file `qe_rcf.v`.

Part I

Infrastructure

1 Booleans in Coq logic

In this chapter, we explain the foundations on which our work lies. We explain what small scale reflection is, and we illustrate its practical use in Coq through the basic definitions from the SSReflect library. The SSReflect library was initially developed to build a formal proof of the Four Colour Theorem [41], and has been progressively extended in order to formalize the Feit-Thompson Theorem [42].

The main ingredient is the type `bool` of Booleans, which is defined as an non recursive inductive type:

```
Inductive bool : Type := true | false.
```

It has two constructors with no arguments, and is hence inhabited by exactly two elements: `true` and `false`.

In Section 1.1, we recall what Boolean reflection is and give an intuitive interpretation for it. Section 1.2 explains what we get by phrasing a Boolean statement rather than a propositional one. In Section 1.3 we give example of practical use of the Boolean reflection in our development.

1.1 Reflection

In the Coq proof assistant, Reflection is a methodology that consists in replacing multiple deductive steps (*i.e.* applications of deduction rules of the Calculus of Construction) by computational steps.

Reflection needs a type of reified objects to compute on, we call this type the *reflected type* or the *reified type*. Usually this latter type is an inductive which constructors represent the construction we want to reflect. For example, if we want to reflect arithmetic, there could be a constructor for zero, a constructor for one, a constructor for the addition and a constructor for the multiplication. We also need an interpretation function for evaluating elements of the reflected type into the initial type. For example, the reified multiplication constructor of reified natural arithmetic should be mapped to the Coq function that multiplies two natural numbers.

Large Scale Reflection

This methodology was introduced by [17] as a way to build efficient decision procedures. The most famous examples of application of this methodology are the `ring` tactic [17, 48], which implements a reflexive decision procedure for ring expressions, the `micromega` tactic [10] for linear arithmetic and the tactic for rewriting modulo AAC [86]. We call this *large scale reflection* because it replaces several deduction steps by a single conversion rule.

Usually, large scale reflection is performed in two steps, as shown in Figure 1.1: the first step is *reification*, in order to transform a term into its reified syntactic tree (which interpretation reduces to the initial term). The second step is the application of the reflexive procedure on this tree, which when interpreted reduces to a provably equal term. However, the user only witnesses the progression of the initial type.

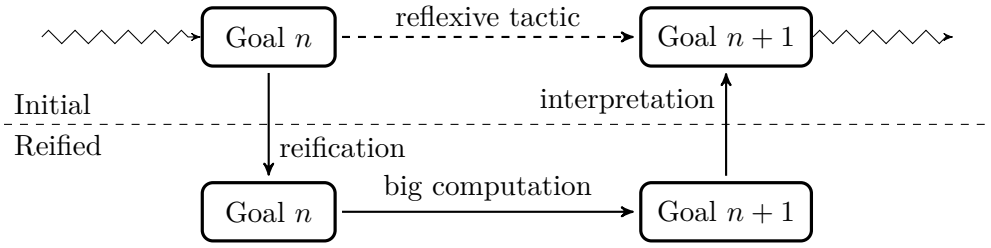


Figure 1.1: Large scale reflection

Let us give an example for the `ring` tactic. Let `R` be a ring, which we call is the initial type. Now, for `ring`, there are two reified types, one for uninterpreted ring expressions (`PolExpr`) and one for normalized uninterpreted ring expressions (`Pol`). We show in Figure 1.2 that the expression `(x * 1)` is reified into the uninterpreted ring expression corresponding to the polynomial $X_1 \cdot 1$, where the indeterminate X_1 is associated to the value x. It is then normalized to the (normalized) uninterpreted ring expression X_1 and reinterpreted as x.

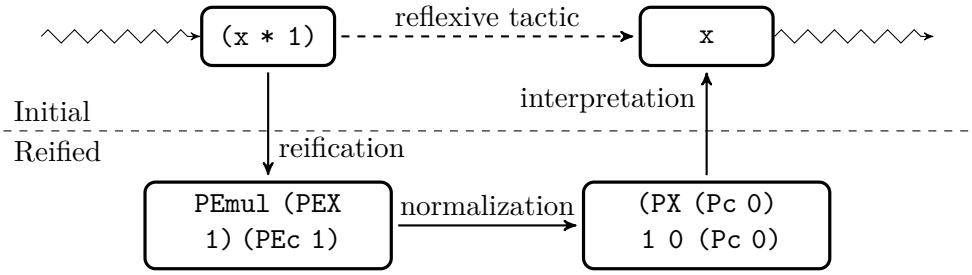


Figure 1.2: The ring tactic

Small scale reflection

The need for an initial type and a reified type is also true for small scale reflection, but instead of having a big and atomic computational reduction of the reified expression, we do manual rewriting or small reduction steps. In this design, the user

controls the progression of the proof goal by interleaving these small transformations with deductive steps, as shown in Figure 1.3

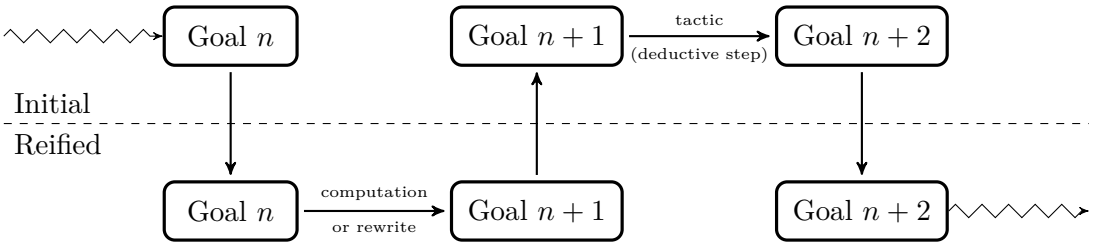


Figure 1.3: Small scale reflection

Boolean reflection (see Figure 1.4) is the process by which we reflect a propositional statement into an equivalent Boolean statement. It is a particular case of small scale reflection where the interpretation function is `is_true`.

```
Definition is_true (b : bool) : Prop := (b = true).
```

This function interprets Booleans as propositions by comparing them to `true`.

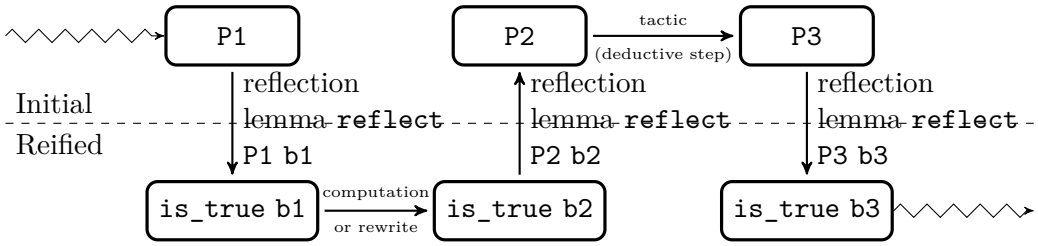


Figure 1.4: Boolean reflection

For example, the standard library distributed with the Coq system defines the comparison of natural numbers as an inductive binary relation:

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : le n n
  | le_S : forall m : nat, le n m -> le n (S m)
```

where the type `nat` of natural numbers is itself an inductive type with two constructors:

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

The proof of `(le 2 2)` is `(le_n 2)` and the proof of `(le 2 4)` is `(le_S (le_S (le_n 2)))`. With this definition of the `le` predicate, a proof of `(le n m)` actually necessarily boils down to applying a number of `le_S` constructors to a proof of the reflexive case obtained by `le_n`. The number of piled `le_S` constructors is exactly the difference between the two natural numbers that are compared.

In the SSReflect library on natural numbers, the reflected counterpart of this definition is a Boolean test:

```
Definition leq (m n : nat) := is_true (m - n == 0).
```

where `(_ == _)` is a Boolean equality test and `(_ - _)` is the usual subtraction on natural numbers. Note that when `n` is greater than `m`, the difference `(m - n)`

is zero. In this setting, both a proof of `(is_true (leq 2 2))` and a proof of `(is_true (leq 2 4))` consist in evaluating this comparison function and check that the output value is the Boolean `true`: the actual proof term is in both cases `(refl_equal true)` where `refl_equal` is the Coq constructor of proofs by reflexivity. However, the principal motivation for small scale reflection is not the reduction of the size of proof terms. Small scale reflection consists in designing the objects of the formalization so that proofs benefit from computation and therefore relieve the user from part of the otherwise explicit reasoning steps.



Remark

In fact, `is_true` is declared as a coercion. This means that there is no need to put it in by hand. The system will apply it automatically whenever the type constraints inferred a element of type `Prop` was required but got a Boolean instead.

The way Boolean reflection is achieved in SSReflect is through the use of the `reflect` predicate.

```
Inductive reflect (P : Prop) : bool -> Set :=
  | ReflectT of  P : reflect P true
  | ReflectF of ~ P : reflect P false.
```



Technical remark

In SSReflect the "`of P`" syntax is a shorthand for "`(_ : P)`".

The inductive type `reflect` is a two constructor inductive that relates the provability of a proposition with `true` and the provability of its negation with `false`. In order to inhabit such an inductive, one has to decide whether the proposition `P` is `true` or `false`.

In the previous example, the reflection lemma is stated like this:

```
Lemma leP m n : reflect (le m n) (leq m n).
```

It can be read (and it is logically equivalent to) the equivalence:

```
(le m n) <-> (leq m n)
```



Remark

Although this reflection lemma `leP` is defined in the `ssrnat.v` file of the SSReflect library, it is almost never used. Indeed, there is no advantage of using `le` instead of `leq`.
However, this is not true anymore for equality, where both forms are useful. We will detail this in Section 1.3.

1.2 Excluded middle and proof irrelevance

In the constructive type theory implemented by the Coq system, the excluded middle principle does not hold in general for any statement expressed in the `Prop` sort. It corresponds to the fact that a proposition might be provable, that its negation might be provable or that neither are provable. An advantage of Boolean statement is that they enjoy excluded middle (a Boolean is either `true` of `false`).

Propositions that can be reflected to Booleans are exactly the decidable propositions. Hence, Boolean reflection is a great tool to ease case analysis, since a `case` on a Boolean-reflected proposition creates two subgoals according to the possible truth values of this Boolean. The SSReflect library provides more elaborated constructs, especially to take advantage of both the Boolean and the predicate it reflects. We give examples of this in Section 1.3.

In Coq, two proofs of the same statement have no reason to be equal. Indeed, equality is intentional, which means it distinguishes two programs that compute the same value (*i.e.* that are extensionally equal), but that are not written in the same way. A traditional example would be that although the bubble sort and the quick sort both sort lists, they are not equal. However, all the proofs of equality of two given Booleans are the same.

```
Lemma bool_irrelevance (x y : bool) (E E' : x = y) : E = E'.
```

In the SSReflect library, this is presented as a consequence of a more general fact: proofs of equality are all equal on types on which equality is decidable.

```
Theorem eq_irrelevance (T : eqType) x y :
  forall e1 e2 : x = y :> T, e1 = e2.
```

This property is called proof irrelevance, [52] we only showed it applies to proof of equalities in an `eqType`, *i.e.* a type on which equality is decidable. The structure of `eqType` is described in Section 2.2. This implies there is exactly zero or one proof of such equality statements: either the two elements are equal, and there is exactly one proof of equality, or they are not, and there is no proof of equality.

In particular, this implies a strong property on the Σ-type `sT := {x : T | P x}` of elements of a type `T` that satisfy a Boolean property `P`: two elements of `sT` are equal if and only if their first projection (*i.e.* the projection `projT1 : sT -> T`) are equal. This is not true in general where two elements of the Σ-type originating from the same element in the type `T` can be different because their proof of membership are. We call *subtypes* the Σ-types for which this property holds.

1.3 Interaction between bool and other types

Discrete types

The most used Boolean operator in the library is the equality operator on types with decidable equality (also called discrete types) `eqType`. Indeed, it reflects Coq intentional equality to a Boolean predicate `eq_op`, which is denoted using the notation `(_ == _)`. Chapter 2 explains how `eqType` is defined and integrated in a larger context.

One of the most simple datatypes on which such a decidable operator can be defined is `bool`. But equality is also decidable on `nat`, the type of Coq natural numbers. This leads to the definition of the `eqn` equality operator on `nat`.

```
Fixpoint eqn m n {struct m} :=
  match m, n with
  | 0, 0 => true
  | m'.+1, n'.+1 => eqn m' n'
  | _, _ => false
```

```
  end.
```

This operator reflects the intentional equality in the sense the lemma `eqnP` is provable:

```
Lemma eqnP : forall x y : nat, reflect (x = y) (eqn x y).
```

Inductive families

There are multiple specific inductive types that play a role similar to `reflect`, in which they associate the provability of a proposition or a list of propositions, with another datatype. One of the most used is the `leq_xor_gtn` inductive family.

```
Inductive leq_xor_gtn m n : bool -> bool -> Set :=
  | LeqNotGtn of m <= n : leq_xor_gtn m n true false
  | GtnNotLeq of n < m : leq_xor_gtn m n false true.
```

And it is used through its only instantiation:

```
Lemma leqP m n : leq_xor_gtn m n (m <= n) (n < m).
```

When we make a case analysis on a proof of `(leqP m n)`, it generates two subgoals, one for each constructor. We explain how this works in Section 4.2, as we reuse this concept and extend it.

Container datatypes

The SSReflect library also provides support for the theory of container datatypes. Those are types equipped with Boolean characteristic functions, and modeled by the structure of `predType`. Any inhabitant of a type equipped with a structure of `(predType T)` should be associated with a total Boolean unary operator of type `(T -> bool)`. This Boolean operator benefits from a generic `(_ \in _)` infix notation: it is a membership test. For instance, if `T` is a type with decidable equality, the type `(seq T)` of finite sequences of elements in `T` has a structure of `(predType T)`, whose membership operator is the usual membership test for sequences. For any sequence `(s : seq T)`, the Boolean expression `(x \in s)` tests whether `x` belongs to the elements of the sequence `s`. A `predType` structure however does not imply the effectiveness of the comparison between its elements: the subset relation between `(a : T)` and `(b : T)`, denoted by `{subset a <= b}`, is not a Boolean test, even if `T` is an instance of `predType`, as there is a priori no effective way to test this inclusion.

For a further introduction to small scale reflection and to the support provided by the SSReflect tactic language and libraries, one may refer to [44].

2 Algebraic Hierarchy

When doing mathematics, the objects we handle may share some structure, for example we can recognize many examples of groups, rings and fields in the literature. In fact, there is a huge number of different structures which one can identify and which all have their own theory. When formalizing mathematics, it is very important to implement the sharing of mathematical structure, because we do not want to keep repeating the same proofs all over the developments. In programming languages, sharing is often achieved by abstraction: we derive all we need from basic facts, and then we specialize to our situation. Moreover, several mathematical structures may share more or less properties, which means the code has to be modular. For example, rings inherit all the properties of commutative groups, so rings should share some theory with commutative groups, which we do not want to duplicate. The purpose of an *algebraic hierarchy* is to organize mathematical structures and their theories in order to maximize sharing.

In this Chapter, we describe the constitution of the SSReflect algebraic hierarchy, essentially as it was made and described in [36]. A huge part of our work (cf Chapter 4) is based on this hierarchy, and reuses the same methodology to achieve the task of building an extension of this hierarchy.

We introduce a lot of important vocabulary in this chapter. In Section 2.1 we explain what we mean by *axiom*. Section 2.2 explains the different structures the SSReflect library contains and what choices of operators, relations and axioms have been made. Section 2.3 gives a short insight on how the structure information is retrieved when needed.

2.1 On the meaning of *axiom*

One very important thing is to clarify first the use of the word *axiom*. In model theory and logic (as in [54, 30]), the axioms are the formulas from which we derive a theory using the rules of the logic (we describe this more formally in Section 10.1). In Coq, the word axiom can refer to the vernacular command `Axiom`, which is used to add an axiom to Coq itself. However, we use the word axiom to describe a property of an interface. Unlike for Coq axioms, we can and we must provide a proof of the axiom when we define a structure. For example, in Section 1.3 we provide a proof `eqnP` of the axiom `eqP` of the interface for discrete types, which is described

in Section 2.2 and represented in Figure 2.1.



Remark

When used inside a module type, the key word `Axiom` takes a different meaning and becomes a statement that has to be satisfied to implement the module, which is relatively close to our use.

In our work, we never make use of Coq axioms. Hence, from now on, the word "axiom" always refers to properties of structures.

2.2 Choice of interfaces

In SSReflect various algebraic structures are represented by *records*, *i.e.* elements of record types. In Coq *record types* are implemented using an inductive type with one constructor. The arguments of this constructor are named the *fields* of the record and the projections are the functions that returns the fields of a record. Their type is hence: "record type" → "type of the field".

In the specific case where the record type characterizes an algebraic structure, we call this record type an *interface*, and the elements of an interface the corresponding *structures*. This differs a little bit from Coq terminology where the vernacular command `Structure` is a synonym for `Record`: we use it only for records that have an algebraic meaning.

In the SSReflect library, structures have two fields: the *carrier*, whose type is `Type` and the *class* of the structure. The class is a record which must contain both the instantiation of the signature and proofs of the axioms of a class. The *signature* is the name and type of the operators and relations, and the axioms are formulas that explain the properties of the operators and characterizes the relations between them. The inheritance of interfaces is implemented by dividing the class in two parts: the first part is the class from the super-interface, while the second part, called *mixin*, is a record that contains the new operators and relations and the proofs of the new axioms. A complete description of the packaging of algebraic structure is given in [37, 36].

We give an example with the four first layers of the algebraic hierarchy in Figure 2.1. The Figure 2.2 zooms out and gives a snapshot of the principal structures we use from the algebraic hierarchy. We then regroup in Figure 2.3 the signature that each new mixin adds to the previous ones, together with some useful definitions.

All the instances of these structures are **discrete**, *i.e.* based on a type with decidable equality. The operator for the equality decision procedure is denoted by `(_ == _)`, and satisfies the following property:

```
Lemma eqP : forall (T : eqType) (x y : T), reflect (x = y) (x == y).
```

This axiom asserts that given two terms of type `T`, the equality decision procedure outputs true if and only if the two terms are equal (for the – so called – Leibniz equality, also known as propositional equality).

They are also equipped with a choice operator `xchoose` of type:

```
Definition xchoose : forall (T : choiceType) (P : T -> bool),
  (exists x, P x) -> T.
```

which satisfies the two following properties:

```
Lemma xchooseP : forall (T : choiceType) (P : T -> bool)
  (xP : exists x, P x), P (xchoose T P xP).
```

```
Lemma eq_xchoose : forall (T : choiceType) (P Q : T -> bool)
  (xP : exists x, P x) (xQ : exists x, Q x),
  (forall x, P x = Q x) -> xchoose T P xP = xchoose T Q xQ.
```

These properties respectively ensure the correctness and uniqueness of the chosen element with respect to the predicate `P`.

For instance, in Coq, any countable type can be provably equipped with such a structure. This means we can take T to be the type $\mathbb{Q}$ of rational numbers.

The choice structure is fundamental to formalize both the comparison of Cauchy reals in Section 5 and the construction of the effective quotient type in Section 7. However it is not crucial for quantifier elimination (Part III).

The `zmodType` structure of commutative groups comes with a number of notations related to the additive notation of a commutative law, including those for iterated additions. The term `x *+ n` denotes `(x + ... + x)` with `n` occurrences of `x`. Non constant iterations benefit from an infrastructure devoted to iterated operators (see [8]) and from a LaTeX-style notation allowing various flavors of indexing: `(\sum_(i <- r) F i)` sums the values of `F` by iterating on the list `r`, while `(\sum_(i \in A) F i)` sums the values of `F` belonging to a finite set `A`, or `(\sum_(n < i <= m | P i) F i)` the values of `F` in the range $]$`n`, `m`$]$ which moreover satisfy the Boolean predicate `P`, etc... This infrastructure also provides a corpus of lemmas to manipulate these sums and split them, reindex them, etc.

The `ringType` structure of nonzero rings inherits from the one of commutative group (and of its notations). In addition, it introduces notations for the multiplicative law, including those for iterated products. The term `x ^+ n` denotes `(x * ... * x)` the exponentiation of `x` by the natural number `n`. Again, we benefit here from the infrastructure for iterated operators: `(\prod_(i <- r) F i)` is the product of the values taken by the function `F` on the list `r`, etc. The infrastructure provides the theory of distributivity of an iterated product over an iterated sum. Finally, a ring structure defines a notation for the canonical embedding of natural numbers in any ring: `n%:R` denotes `(1 + ... + 1)`.

The `ringType` structure has variants respectively for commutative rings, rings with units (*i.e.* where it is decidable for an element to be invertible), commutative rings with units and integral domains. A field is a commutative ring with units in which every nonzero element is a unit.

Finally, scaling operations are available in module structures: a left module provides a left scaling operation denoted by `(_ *: _)` and a left algebra structure defines an embedding of its ring of scalars into its carrier: `(fun k => k *: 1)`. The `algType` (resp. `unitAlgType`) structure of algebra equips rings (reps. rings with units) with scaling that associates both left and right.

The `decFieldType` structure equips fields with a decidable first-order equational theory by requiring a satisfiability decision operator for first-order formulas on the language of rings.

The `closedFieldType` structure equips algebraically closed fields. It inherits from the `decFieldType` structure: a structure of algebraically closed field has to be built on a decidable field. This may disturb at first glance since the first-order theory of algebraically closed field enjoys quantifier elimination and is hence decidable.

This design choice in fact allows the user to specify explicitly the preferred decision procedure, which might not be quantifier elimination, for example in the case of finite fields.

2.3 Structure inference

In SSReflect

When we consider elements of an algebraic structure, we implicitly refer to the elements of the carrier of the structure. For example, given a ring `(R : ringType)`, taking an element `(x : R)` is in fact taking `(x : Ring.sort R)`, where `Ring.sort` is the projection from the record to the carrier and is inserted implicitly by the coercion mechanism.

So for example, writing `forall x y : R, x * y = y * x` is typable and the result is expanded to the following form, when removing notations and showing implicit arguments.

```
forall x y : Ring.sort R, @mul R x y = @mul R y x
```

because the multiplication operator has the following signature:

```
Definition mul : forall R : ringType, R -> R -> R
```

So far, there is no need for structure inference. Problems come when one does not use directly the appropriate structure. For example integers, as defined in Section 4.4.1, have a ring structure. We should be able to write `forall x y : int, x * y = y * x`, but `int` has type `Type`, not `ringType` (the ring structure of `int` was named `int_Ring`). Coq provide a mechanism to help the unification to find a solution. It is called *canonical structure* [77]. Indeed, the Coq system has to type:

```
forall x y : int, @mul ?ringType x y = @mul ?ringType y x
```

In this term, $?_{\texttt{ringType}}$ is a meta-variable which is inserted automatically (through the implicit arguments and notation mechanism) and which has to be resolved by the unification algorithm of Coq.

Since `x` and `y` should have type (`Ring.sort` $?_{\texttt{ringType}}$) but have type `int`, this triggers a unification problem of the form:

$$\texttt{Ring.sort}\ ?_{\texttt{ringType}} \equiv \texttt{int}$$

The structure `int_Ring` is the solution to this problem. By registering this canonical solution using the `Canonical` vernacular command, the unification algorithm will look for an entry corresponding to the pair (`Ring.sort`, `int`) in the canonical structure table and will find the entry `int_Ring`.

The command to print the canonical structure table in Coq is `Print Canonical Projections` and outputs results in the form:

```
key <- projection ( canonical structure )
```

So for our example with `int` being canonically a `ringType`, this gives:

```
int <- Ring.sort ( int_Ring )
```

The same problem occurs with inheritance. For example, addition has the following signature:

```
Definition add : forall Z : zmodType, Z -> Z -> Z
```

If we want to write `(forall x y : R, x + y = y + x)` where `R` still has type `ringType`, the system should reject the statement, because it should not be typable. When we remove the notations and put question marks in places where the system should be able to fill in the information through type inference, we get

```
forall x y : Ring.sort R, @add ?zmodType x y = @add ?zmodType y x.
```

Here, the unification algorithm encounters the unification problem:

$$\texttt{Zmodule.sort}\ ?_{\texttt{zmodType}} \equiv \texttt{Ring.sort R}$$

But each `zmodType` has been declared has canonically a `ringType`, using the canonical structure `Ring.zmodType`. So the pair (`Zmodule.sort`, `Ring.sort`) was added to the canonical structure table and corresponds to the line

```
Ring.sort <- Zmodule.sort ( Ring.zmodType )
```

when using the command `Print Canonical Projections`.

The search for the appropriate structure is done at the core of the unification algorithm, which makes it very fast.

Comparison with other libraries tools and systems

The math classes development by van der Weegen, Spitters and Krebbers [81, 87] also builds a hierarchy of interfaces. There are two fundamental differences between it and the SSReflect one. The first difference is that it is defined in category theory and does not give much importance to the decidability of predicates such as equality or comparison, and it does require a choice operator, as the SSReflect algebraic hierarchy does.

The other difference is that the inference mechanism is based on type classes [80] rather than canonical structures [77]. In Coq, type classes where designed to solve the same kind of problems as canonical structures. Structure definition and inference using type classes is more flexible than with canonical structures in that they allow for multiple possible instances and backtracking. They use proof automation instead of being hard-wired into the unification algorithm, as described in [80], which is probably what makes it less efficient in practice.

In Matita [1], unification hints [2] is also a mechanism to manually help the unification to find solutions in case of failure, but in a more fine-grained way than canonical structures. They are even more flexible than type classes, in the way they let the user list the problems and their solution by hand. Combined with a good meta-language, they could be used to encode both type classes and canonical structures.

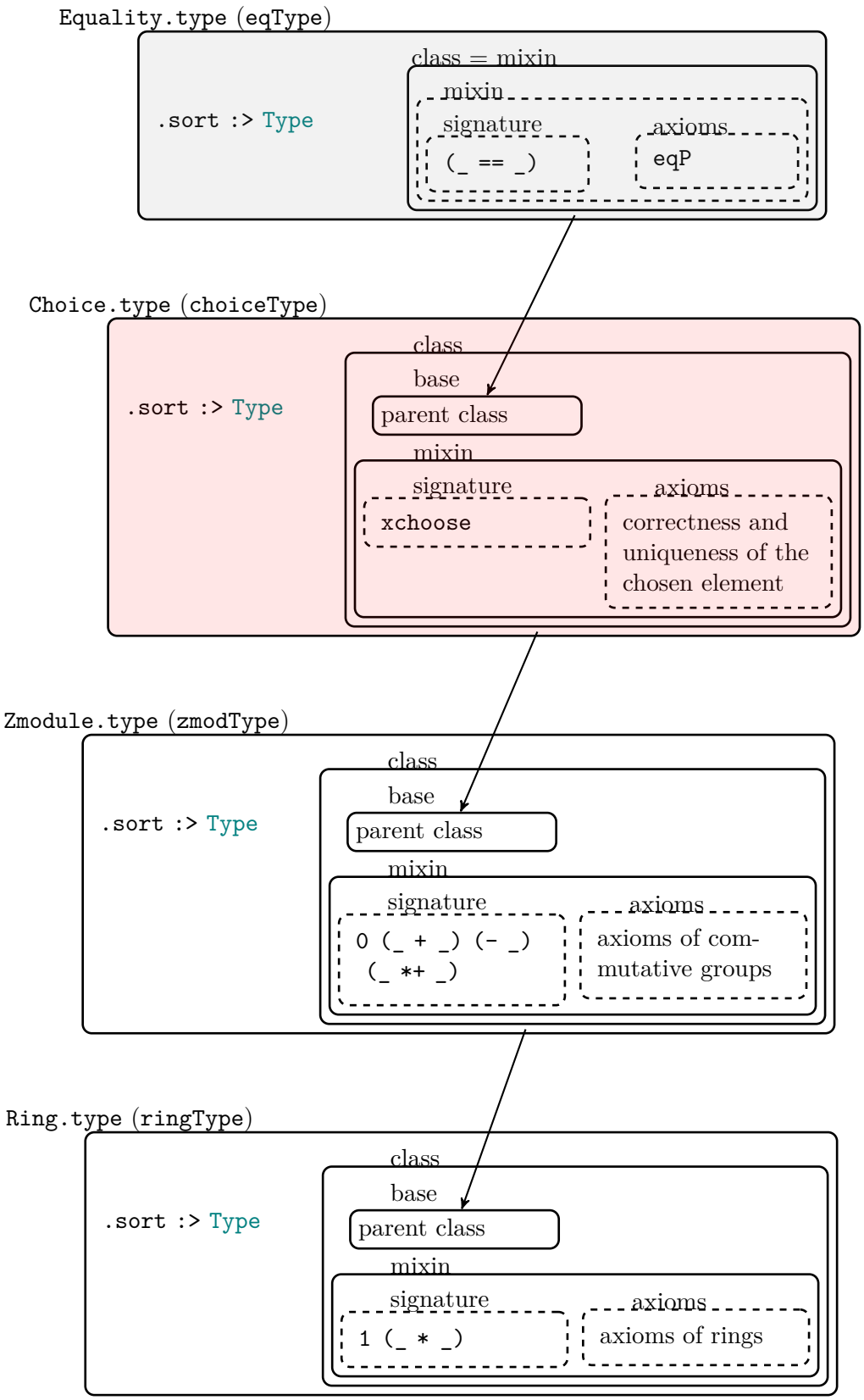


Figure 2.1: Packaging of mathematical structures

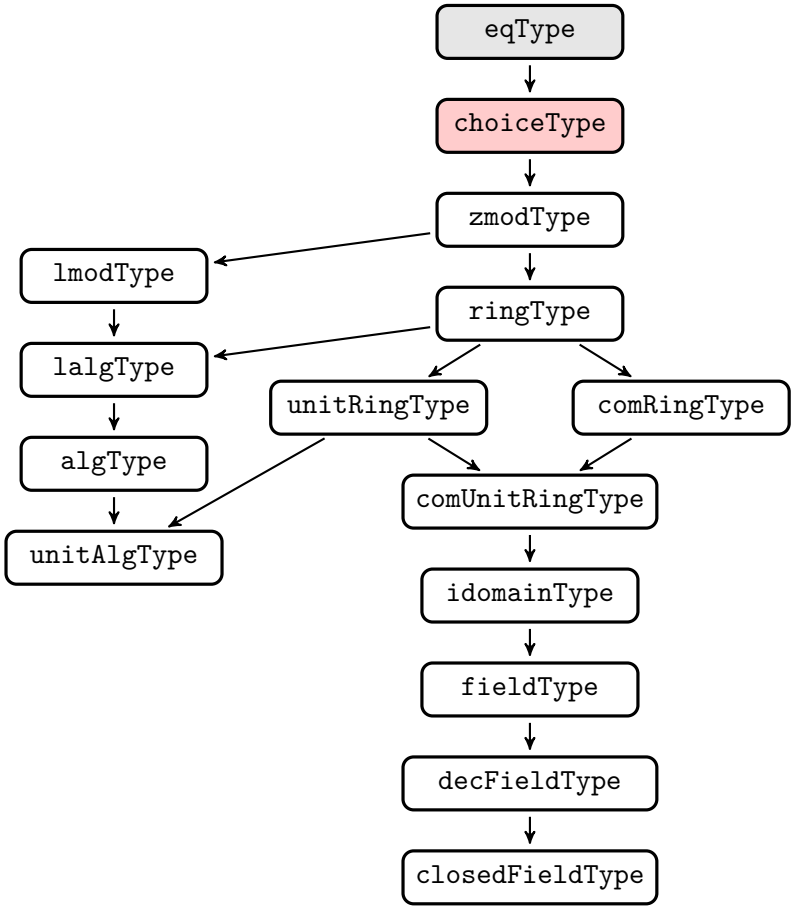


Figure 2.2: The algebraic hierarchy

Name of the structure	Description	Signature additions Useful definitions	Notation
`eqType`	Type with decidable equality	Boolean equality test of `x` and `y`	`x == y`
`choiceType`	Type with choice	choice operator	`xchoose`

`zmodType`	Commutative group	additive identity
		addition of `x` and `y`
		`n` times `x`, with `n` in nat
		opposite (additive inverse) of `x`
			`0`
			`x + y`
			`x *+ n`
			`- x`

		difference of `x` and `y`	`x - y`
		opposite of `x *+ n`	`x *- n`
		iterated sum	`\sum_<range> e`
		`i`-th element of the sequence `l` with default value `0`	`l`_i`
`ringType`	Ring	multiplicative identity ring product of `x` and `y`	`1` `x * y`
		ring image of `n`, with `n` in nat	`n%:R`
		iterated product	`\prod_<range> e`
		`x` to the `n`-th power with `n` in nat	`x ^+ n`
`unitRingType`	Ring with units	test of invertibility of `x` ring inverse of `x`, if x is a unit, else x	`x \in GRing.unit` `x^-1`
		`x` divided by `y`, *i.e.* `x * y^-1`	`x / y`
		inverse of `x ^+ n`	`x ^- n`
`lmodType R`	Left module on the scalar ring `R`	`v` scaled by `a` an element of the scalar ring	`a *: v`
`lalgType R`	Left algebra on the scalar ring `R`	image of the scalar `k` in a left algebra	`k%:A`

Figure 2.3: Signatures of SSReflect algebraic structures

3 Polynomials

This chapter is devoted to the description of polynomials in the SSReflect library. Polynomials are one of the major ingredients in almost all our developments: algebraic numbers are roots of polynomials, real and algebraically closed field are defined in terms of polynomials and quantifier elimination on their theory is based on polynomial root counting.

Section 3.1 describes the representation of polynomials and a few operation on them: ring operations, euclidean division, greatest common divisors and relative primality. It also describes the concept of pseudo-division [5]. In Section 3.2, we recall what a resultant is, and we exhibit the properties we need from it. We also explain how to build and handle bivariate polynomials from univariate polynomials. Although the idea seems natural, their manipulation ends to be quite disturbing.

3.1 Polynomial arithmetic

Representation and operations

SSReflect represents univariate polynomials as lists of coefficients with lowest degree coefficients in head position. Hence, the coefficient of X^i of a polynomial is the i^{th} element of the list (starting from 0). We require polynomials to be in normal form, in the sense that the last element of the list is never zero. Hence the type `{poly T}` of polynomials with coefficients in the type `T` is a Σ-type, which packages a list and a proof that its last element is nonzero. The zero polynomial is therefore represented by the empty list.

It is convenient and standard to define the degree of a univariate monomial as its exponent, except for the zero constant, whose degree is set at $-\infty$. Then the degree of a polynomial is the maximum of the degree of its monomials. To avoid introducing option types, we simply work here with the size of a polynomial, which is the size of its list. This lifts the usual codomain of degree from $\{-\infty\} \cup \mathbb{N}$ to $\mathbb{N}$ since in our case:

$$\text{size}(P) = \begin{cases} 0 & \text{, if and only if } P = 0 \\ \deg(P) + 1 & \text{, otherwise.} \end{cases}$$

The advantage of using the size instead of the degree is that we only use natural

number operations and the library on natural numbers is quite complete. However, we lose the fact that the degree was a semi-ring morphism, which makes lemmas stated using size quite unnatural.

For example, let us take the function `size_proper_mul`, defined as follows

```
Lemma size_proper_mul P Q : lead_coef P * lead_coef Q != 0 ->
  size (P * Q) = (size P + size Q).-1.
```

where `.-1` is a unary operator for the predecessor function on natural numbers, with the convention that `0.-1` reduces to `0`.

There is a `.-1` in the conclusion and a side condition on `P` and `Q`. In our experience, these side conditions introduce quite a lot of overhead and maybe the theory of polynomial degree would gain by using a proper degree function together with a type for $\{-\infty\} \cup \mathbb{N}$.

Polynomials are equipped with a ring structure, taking advantage from all the definitions and all the theory of rings. If the base type has more structure, the polynomial ring might also have more structure: for example, polynomials over a commutative ring are commutative and polynomials over an integral domain also form an integral domain.

The SSReflect library also defines an evaluation method for polynomial, using Horner's algorithm. The evaluation of the polynomial `P` in an element `x` is denoted `P.[x]`. Roots of polynomials are characterized using the predicate `root` defined as follows.

```
Definition root (R : ringType) (P : {poly R}) (x : R) := (P.[x] == 0).
```

Pseudo division and greatest common divisor

When R is an integral domain, it is no longer possible in general to program the Euclidean division algorithm on $R[X]$ as it would be if R was a field. The usual polynomial Euclidean division actually involves exact divisions between coefficients of the arguments, which might not be tractable inside R. However it might still remain doable if the dividend is multiplied by a sufficient power of the leading coefficient of the divisor. For instance one cannot perform Euclidean division of $2X^2+3$ by $2X+1$ in $\mathbb{Z}[X]$, but one can divide $4X^2+6$ by $2X+1$ inside $\mathbb{Z}[X]$. In the context of integral domains, Euclidean division should be replaced by *pseudo-division*.



Definition: Pseudo-division

> Let R be an integral domain. Let P and Q be elements of $R[X]$. A *pseudo-division* of P by Q is an Euclidean division of αP by Q, where α is an element of R such that Euclidean division can be performed inside $R[X]$.

Note that α always exists and can be chosen to be a sufficient power of the leading coefficient of Q. We implement a pseudo-division algorithm, which given two polynomials `P` and `Q`, computes three results: a natural number (`rscalp P Q`) such that (`lead_coef Q ^+ (rscalp P Q)`) is a sufficient α, a polynomial (`rdivp P Q`), the corresponding pseudo-quotient, and a polynomial (`rmodp P Q`), the corresponding pseudo-remainder. They satisfy the following specification:

```
Lemma rdivp_eq P Q :
  (lead_coef Q ^+ (rscalp P Q)) *: P = (rdivp P Q) * Q + (rmodp P Q).
```

Each possible value of α leads to different values for the pseudo-quotient (resp. pseudo-remainder) of two polynomials, but they are always associated. We say that `P` *pseudo-divides* `Q`, denoted `(P %| Q)` if the pseudo-remainder of `P` by `Q` is zero. We recover some standard lemmas about divisibility like:

```
Lemma dvdp_mul : forall D1 D2 M1 M2 : {poly R},
  D1 %| M1 -> D2 %| M2 -> D1 * D2 %| M1 * M2.
```

The pseudo greatest common divisor `rgcdp` is obtained by replacing division by pseudo-division in the Euclidean algorithm. This is not the optimal algorithm to compute such a greatest common divisor, which is a non trivial problem. We choose here a naive implementation, since at this point, we are not concerned with efficiency. However we recover standard properties of the greatest common divisor, like:

```
Lemma root_rgcd : forall P Q x,
 root (rgcdp P Q) x = root P x && root Q x.
```

We denote by $\mathrm{gcdp}_{i=1}^{n} P_i$ the iteration of the `rgcdp` function on the list $(P_i)_{i=1\ldots n}$ (with at least two elements). The polynomial $\mathrm{gcdp}_{i=1}^{n} P_i$ has a root at x if and only if x is a common root of the $(P_i)_{i=1\ldots n}$.

3.2 Resultant

The *resultant* of two polynomials $P = \sum_{i=0}^{m} p_i X^i$ et $Q = \sum_{i=0}^{n} q_i X^i$ is usually defined as the determinant of the Sylvester matrix, which is a square $(m+n)$ matrix.

$$\mathrm{Res}_X(P,Q) = \begin{vmatrix}
p_m & p_{m-1} & \cdots & p_{j+1} & p_j & \cdots & p_1 & p_0 & 0 & \cdots & 0 \\
0 & \ddots & \ddots & & & & \ddots & \ddots & & \ddots & \vdots \\
\vdots & \ddots & \ddots & \ddots & & & & \ddots & \ddots & & 0 \\
0 & & 0 & p_m & p_{m-1} & \cdots & p_{j+1} & p_j & \cdots & p_1 & p_0 \\
q_n & q_{n-1} & \cdots & q_1 & q_0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & \ddots & \ddots & & \ddots & \ddots & \ddots & & & & \vdots \\
\vdots & \ddots & \ddots & \ddots & & \ddots & \ddots & \ddots & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & \ddots & \ddots & \ddots & & \ddots & \ddots & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & & \ddots & \ddots & 0 \\
0 & \cdots & \cdots & \cdots & \cdots & 0 & q_n & q_{n-1} & \cdots & q_1 & q_0
\end{vmatrix}$$

The notion of resultant is well described and studied in numerous books, we invite the reader to look in one of them, for instance in [60]. If the polynomials are univariate the resultant is a scalar, if they are bivariate, it is a univariate polynomial in the remaining variable.

We chose to represent bivariate polynomials by iterating twice the construction of polynomials, *i.e.* using the type `{poly {poly R}}`. The `swapXY` function is used to swap the variables and `sizeY` to get the size using the second variable.

In our development we use only the following two properties of the resultant

$$\mathrm{Res}_X(P(X,Y),Q(X,Y)) \in R[Y]$$

of polynomials $P, Q \in R[X,Y]$ where F is a field:

$$\exists U, V \in R[X,Y], \quad \mathrm{Res}_X(P,Q) = UP + VQ$$
$$\text{and}$$
$$\mathrm{Res}_X(P,Q) = 0 \quad \Leftrightarrow \quad P \text{ and } Q \text{ are not coprime as polynomials in } X$$

which respectively express that the resultant of P and Q is in the ideal generated by P and Q, and is zero if and only if P and Q are not coprime as polynomials in X with coefficients in $R[Y]$, *i.e.* they have no common factor in $(R[Y])[X]$.

Moreover we use the following corollary to Bézout theorem: if P and Q are not coprime as polynomials in X with coefficients in $R[Y]$, there exist U and V in $R[X,Y]$ such that U is nonzero, $\deg_X(U) < \deg(Q)$ and

$$U(X,Y)P(X,Y) = V(X,Y)Q(X,Y).$$



Remark

It is very important to note that those results about bivariate polynomials are seen as univariate polynomials with coefficients in $R[Y]$. In particular if P and Q in $R[X,Y]$ are coprime relatively to X, Bezout theorem states that there are three polynomials $U, V, W \in R[Y]$ such that

$$UP + VQ = W$$

which means that P and Q may not be coprime for instantiations of Y which are roots of W.

Part II

Construction of numbers

4 Numeric rings

The algebraic structures presented in Chapter 2 provide a Boolean operator to test the equality between elements, but none of the interfaces presented include an order relation in its signature. Our goal here is neither to allow for the most general framework nor to study the abstract theory of ordered domains. We focus on modeling algebraic structures to capture the order properties of integer, rational, real algebraic and algebraic numbers. We thus build a hierarchy of interfaces with an order and a norm operator which are compatible with the algebraic laws of the structure. We call such structures and interfaces *Numeric* as they deal with numbers, and because of the analogy with the Haskell terminology [71, 57]. This chapter is devoted to the description of low-level design choices for the construction of these interfaces and their theories, which were mainly driven by the developments contained in this thesis, but also by requirements from the proof of Feit-Thompson Theorem.

4.1 Extending the hierarchy

4.1.1 The Numeric Hierarchy

We extend the algebraic hierarchy described in Section 2.2 and displayed on Figure 2.2 by introducing normed and ordered versions of the discrete integral domain and discrete field structures. This amounts to duplicating the corresponding branches as displayed on Figure 4.1.

We call this extension the numeric hierarchy. It is divided in two parts:

- *Numeric interfaces* are extensions of the interfaces from the algebraic hierarchy using a mixin with three operators: two Boolean comparison functions `(_ <= _)` and `(_ < _)` and a norm operator `` `|_| ``, which are compatible with the ring operations. More precisely, the *Numeric mixin* contains the three operators above and the axioms listed in Figure 4.2. They model respectively *Numeric domains* and *Numeric fields*

- We define a `real` predicate characterizing a number as *real* when it is comparable to zero. Thus, what we call *Real interfaces* in Figure 4.1 are extensions of Numeric interfaces using the axiom which states that any number is real:

```
Definition real_axiom : Prop := forall x : R, x \is real.
```

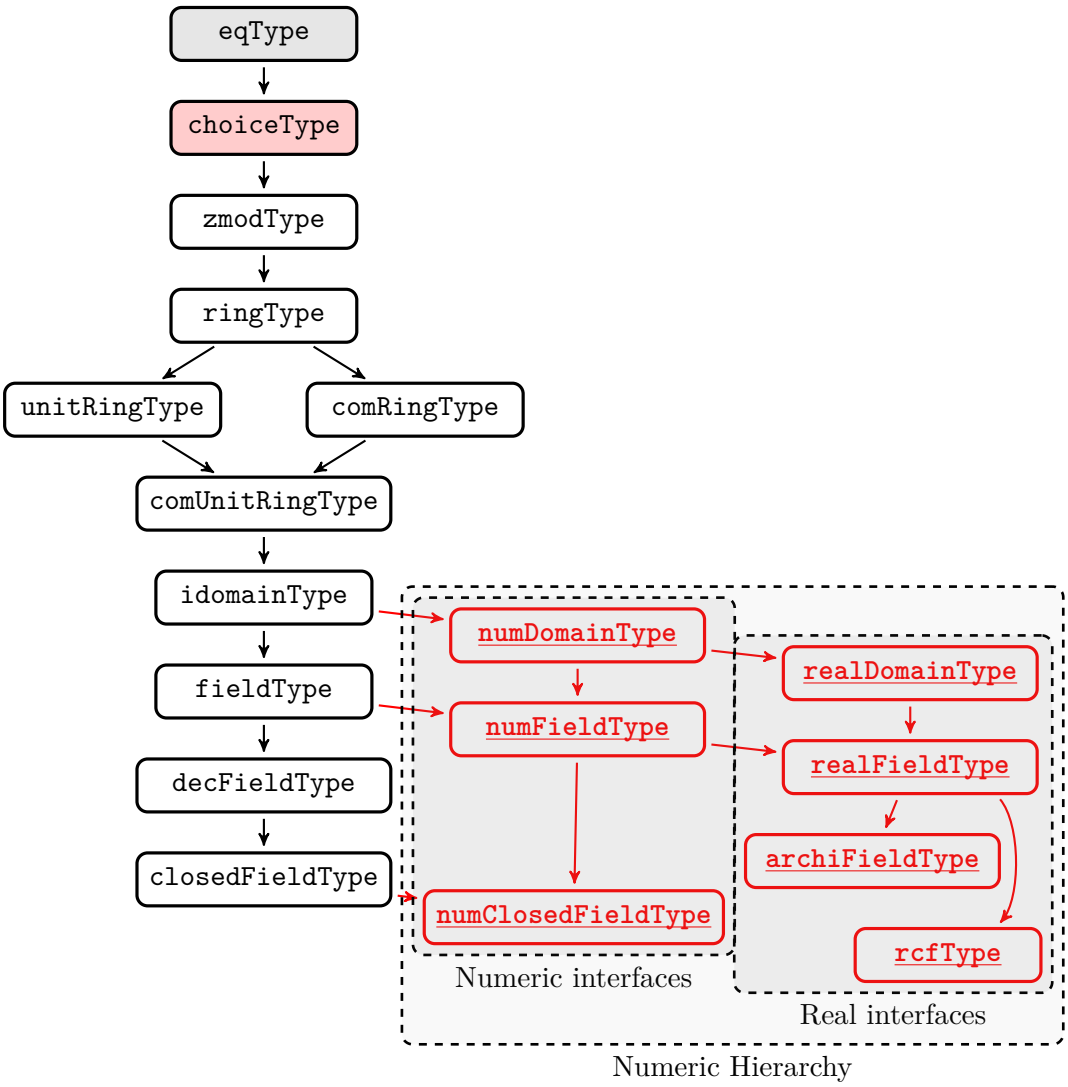


Figure 4.1: Extension of the hierarchy with Numeric and Real interfaces

They model respectively *Real domains* and *Real fields*.

In addition, we provide two extensions of real fields:

– The *Archimedean* (`archiFieldType`) interface adds the Archimedean axiom, which states that any number can be bounded by a natural number:

```
Definition archimedean_axiom : Prop :=
  forall x : R, exists ub : nat, `|x| < ub%:R.
```

– The *real closed field* (`rcfType`) interface adds the real closed field axiom, which states the intermediate value theorem for polynomials with coefficients in the field:

```
Definition real_closed_axiom : Prop :=
  forall (p : {poly R}) (a b : R), a <= b ->
  p.[a] <= 0 <= p.[b] -> exists2 x, a <= x <= b & root p x.
```

```
  (* Triangle inequality *)
forall x y, `|x + y| <= `|x| + `|y|
  (* A sum of positives is positive *)
forall x y, 0 < x -> 0 < y -> 0 < x + y
  (* An element of norm zero is zero *)
forall x, `|x| = 0 -> x = 0
  (* Positive numbers are comparable *)
forall x y, 0 <= x -> 0 <= y -> (x <= y) || (y <= x)
  (* The norm is a morphism for the product *)
forall x y, `|x * y| = `|x| * `|y|
  (* Characterization of <= it terms of the norm *)
forall x y, (x <= y) = (`|y - x| == y - x)
  (* Characterization of < it terms of <= *)
forall x y, (x < y) = (y != x) && (x <= y)
```

Figure 4.2: Axiom of the Numeric mixin

The choice for this particular axiom is further debated in Section 8.1 as we had the choice between four possibilities.

The properties stated in Figure 4.3 are part of the theory of Real domains. But the combination of the Numeric mixin with the `real_axiom` can also be deduced from these properties. As a courtesy, we provide a function that takes those properties and produce both an instance of the Numeric mixin and a proof of `real_axiom`.

```
  (* A sum of positives is positive . *)
forall x y, 0 <= x -> 0 <= y -> 0 <= x + y
  (* A product of positives is positive . *)
forall x y, 0 <= x -> 0 <= y -> 0 <= x * y
  (* An element which is both positive and negative is zero. *)
forall x, 0 <= x -> x <= 0 -> x = 0
  (* Characterization of <= it terms of positive numbers *)
forall x y, (0 <= y - x) = (x <= y)
  (* Any number is comparable to zero *)
forall x, (0 <= x) || (x <= 0)
  (* The norm is invariant by negation *)
forall x, `|- x| = `|x|
  (* Positive numbers are identical to their norm *)
forall x, 0 <= x -> `|x| = x
  (* Characterization of < it terms of <=. *)
forall x y, (x < y) = (y != x) && (x <= y)
```

Figure 4.3: Axioms to produce directly a Real structure

The primitive operation in the signature of the numeric mixin happens to be the norm. Indeed, the order operators can be deduced from it. It is only by commodity that we include the three of them in the signature, as we let the user implement

them as he wishes as long as they satisfy the two last properties of the numeric mixin. Moreover it makes it possible to easily express axioms like the triangular inequality, while if only the norm was available at this stage, it would look like:

```
forall x y, `|`|x| + `|y| - `|x + y|| = `|x| + `|y| - `|x + y|.
```

However, in the real numeric mixin, any of the three operators would be sufficient to define the others, but we still include the three of them in the signature for the same reasons as above.

We now exhibit some data types that we prove are equipped with structures from the numeric hierarchy:

- The data type `int` of integers described in Section 4.4 is equipped with a `realDomainType` structure.
- The data type `rat` of rationals described in Section 4.4 is equipped with an `archiFieldType` structure.
- The data type `(alg rat)` of real algebraic numbers described in Section 7 is equipped with both a `realDomainType` structure and `archiFieldType` structure. So would real algebraic extensions of the rationals be.
- The type `(complex (alg rat))` of algebraic numbers discussed in Section 8.1 is equipped with a structure of `numClosedFieldType`. So would algebraic extensions of the rationals be.

Of course, any data type equipped with a structure is systematically equipped with all the structures located above in the hierarchy. We notice that we do not witness any inhabitant of the `numDomainType` interface which is not also an inhabitant of a higher interface. However, it is still useful to keep this interface, as its theory factorizes the theory of the interfaces depending on it: two thirds of the file dealing with the numeric hierarchy are devoted to the theory of the `numDomainType` interface alone.

Let us remind that our numeric structures integrate a Boolean comparison and a Boolean equality operators. This gives us facilities to carry out our proofs, as described in Section 4.2.

4.1.2 Discussion on the interfaces and their names

In the traditional mathematical terminology, a *formally real field* is a field where (-1) is not a sum of squares. The signature of formally real fields do not contain more symbols that the one of fields. In particular there is no order or norm symbol. Our interface of real field however has an order and a norm in its signature, and its theory shows that squares are positives and that (-1) is negative, so it is no sum of squares. Therefore, any structure of real field in our sense would be a particular instance of a formally real field, by forgetting the extra operators.

Classically, any formally real field can be equipped with a total order which may not be uniquely determined. Such a field equipped with a total order compatible with the algebraic laws of field is usually called a *totally ordered field*. But from a constructive point of view, it is not possible to decide whether an element should be classified as positive or negative. This is the reason why our interface includes a comparison operator that sets once and for all an order on the elements of the field.

We have not created a full-fledged infrastructure for binary relations. Instead we selected precisely the interfaces that abstracted the structures of the data types we used. In other words, we only created interfaces for which we were interested into at least two instances, or interfaces which factored out others. Their main purpose was to work in a unified setting, in which we could put together a common basis of notations and a common battery of lemmas. The choices of abstraction we made in our hierarchy are debatable, but were sufficient to carry out the results we wished for. Moreover, this work did not intend to serve as a classification of the possible axiomatizations containing order and algebraic operations.

For the latter purpose, we refer to the development of *math classes* [81, 87] which establishes a full classification of algebraic structures (with and without order) in a very modular way and expressed in the language of categories. This library has been used in a C-CoRN development for modeling efficient computation on constructive real numbers [59] in order to generalize a previous work [66].

Despite the existence of such an apparently complete classification, we chose not to depend on it, mainly because conventions and choices of design differed irretrievably from the ones that are made in SSReflect. Indeed, the math classes classification and inference mechanism is intensively based on type classes, whereas the SSReflect algebraic hierarchy relies on canonical structures. At the time we conducted our experiments, those two mechanism were not fully compatible yet.

Moreover, the math classes library do not take into account the potential decidability of predicates such as equality, ordering or unit, nor do they include the possibility to have a choice function as the SSReflect hierarchy does. In our developments, those features are nonetheless essential. The price we would pay by adding them to the math classes library would give birth to a whole new classification about the algorithmic of those decidable predicates, and add a lot of theory because of this stronger axiomatic.

In addition to those incompatibilities, the theory of individual structures is not as well developed as we need it to be for our concrete developments. In order to carry out our proofs easily, we need that individual theories provide numerous surgery lemmas so that we do not have to prove well known facts in the middle of a proof of a more elaborated statement.

4.2 Signs, case analysis based on comparisons for reals

The elementary theory of numeric and real domains essentially consists of numerous surgery lemmas describing how ring operations and constants combine with the order relation. We also define for numeric domains the binary operations of minimum, maximum and the unary operations of sign, which make sense only when applied to real numbers. All these definitions are quite standard and do not deserve much comment, except for the sign operation. Indeed, there are several possible choices for its type. One can for instance design a specific inductive type with three constructors to describe the sign of the argument, like the `comparison` type present in the standard library of Coq. However, when we look at the traditional abuses of notations with regard to sign, we observe that it is both used as a element from the numeric domain we are considering or as an integer, and that sometimes the sign of zero is zero and some times it is one. A single sign function is not sufficient to reflect all these behaviours, which is why we define two different sign functions

and provide support for an alternative form for it. The two functions we define for `(R : numDomainType)` are:

```
Definition sgr : R -> R
Definition sgz : R -> int
```

which are zero when their argument is zero. And the alternative form is:

```
(-1) ^+ (x < 0)
```

which can be interpreted as an integer or a element of `R` depending on the context. We also provide lemmas establishing the relations between these forms.

```
Lemma sgr_def x : sgr x = (-1) ^+ (x < 0) *+ (x != 0).
Lemma sgz_def x : sgz x = (-1) ^+ (x < 0) *+ (x != 0).
Lemma sgrEz x : sgr x = (sgz x)%:~R.
```

For instance, one can prove the following result:

```
Lemma sgp_right_scale : forall (c : R) (p : {poly R}) (x : R),
  sgp_right (c *: p) x = sgr c * sgp_right p x.
```

where `R` is a real domain, `p` a polynomial with coefficients in `R`, and `(sgp_right p x)` the sign of the polynomial `p` on the right neighborhood of `x` (see Section 9.2).

A common pattern in proofs involving order is a case analysis on the order between two elements, or on the sign of an expression. Such a case analysis could have two of three branches depending on whether we should consider the case of equality in a specific sub-goal. This pattern is so common that it is important to provide a convenient tool for the user to generate three subgoals whose context are augmented with the sign hypothesis corresponding to each branch of the case analysis. In our context where comparison statements are Booleans, it is always possible to perform a case analysis on the Boolean value of terms of the form `(x <= y)`.

As shown in Figure 4.4, this is not a good option. It indeed generates two subgoals, one with a new hypothesis of type `(x <= y)` and the other with a new hypothesis of type `(x <= y)= false`, whereas in the second case we would have like to get an hypothesis of the form `(y < x)`. Moreover, this style does not allow for a three way case analysis.

This issue can be solved by working with disjunctive statements, even with a `sumbool` type, expressing the possible results of a comparison. This approach needs a little bit of generalization to handle the three way case analysis. Indeed we could define a three constructor inductive type `treeway` to extend `sumbool` to three cases, as shown in Figure 4.5.

However, as shown in Figure 4.6, a case analysis using `threewayP` not replace occurrences of `(x <= y)` by `true` or `false` whereas the naive case analysis on `(x <= y)` did. And it certainly does not replace `(y < x)`, `(y <= x)` and `(x == y)` by the appropriate Booleans, although their values can already be determined.

In the vein of the `reflect` inductive (seen in Section 1.1), or the `leq_xor_gtn` inductive (mentioned in Section 1.3), we define a specific inductive family modeling the specification of a three way case analysis. A slightly simpler variant of this solution had already been proposed by G. Gonthier in the SSReflect library dedicated to natural numbers (which we first directly applied as such [26]). The idea is to relate propositional specifications to Boolean values by an inductive predicate with one constructor per branch of the specification. We relate the simultaneous

Example

```
R : realDomainType
x y : R
P : R -> R -> bool -> bool -> bool -> bool -> R -> Type
============================
P x y (x <= y) (y < x) (y <= x) (x == y) `|x - y|
```

```
case Hxy: (x <= y)
```

```
R : realDomainType
x y : R
P : R -> R -> bool -> bool -> bool -> bool -> R -> Type
Hxy : (x <= y) = true
============================
P x y true (y < x) (y <= x) (x == y) `|x - y|

(* subgoal 2 is : *)
Hxy : (x <= y) = false
============================
P x y false (y < x) (y <= x) (x == y) `|x - y|
```

Figure 4.4: Naive case analysis on an order statement

values of several Booleans with a specification. For example, for the three way case analysis, we define in Figure 4.7 the predicate `comparer` and the lemma `ltrgtP`.

The type `comparer` is an inductive family with indexes which are both Booleans and elements of the real domain `R`, parameterized by two elements of `R`. Each constructor corresponds to a propositional specification: `ComparerLt` corresponds to the specification of the proposition `(x < y)`, `ComparerGt` to the one of `(y < x)` and `ComparerEq` to the one of `(x == y)`. As shown in Figure 4.8, case analysis on an element from the inductive family `comparer`, such as `ltrgtP`, always generates three goals, one by constructor, and adds an extra hypothesis to each goal: the proposition to which the constructor corresponds.

Moreover, the destruction of `(ltrgtP x y)` replaces in each sub-goal all the occurrences of the indexes by the values given in the specification (see Figure 4.8 again). This is of special interest in the case the initial goal `P` contains `(if ... then ... else)` expressions as favored by a Boolean reflection methodology. Of course this solution applies to the two case disjunction by defining a two constructor inductive, respectively specified by `(x <= y)` and `(y < x)`.

Example

```
Inductive threeway (A B C : Prop) : Type :=
  | CaseA of A | CaseB of B | CaseC of C.
```

And then prove:

```
Lemma threewayP x y: threeway (x < y) (y < x) (x == y).
```

Figure 4.5: A three constructor inductive instead of `sumbool`

Example

```
R : realDomainType
x y : R
P : R -> R -> bool -> bool -> bool -> bool -> R -> Type
============================
P x y (x <= y) (y <= x) (x < y) (x == y) `|x - y|
```

```
        case: (threewayP x y)
```

```
R : realDomainType
x y : R
P : R -> R -> bool -> bool -> bool -> bool -> R -> Type
============================
x < y -> P x y (x <= y) (y < x) (y <= x) (x == y) `|x - y|

(* subgoal 2 is *)
y < x -> P x y (x <= y) (y < x) (y <= x) (x == y) `|x - y|

(* subgoal 3 is *)
x == y -> P x y (x <= y) (y < x) (y <= x) (x == y) `|x - y|
```

Figure 4.6: Three way case analysis with a simple inductive

```
Inductive comparer x y :        R ->    R ->
     bool -> bool -> bool -> bool -> bool -> bool -> Set :=
  | ComparerLt of x < y : comparer x y (y - x) (y - x)
     false   false   true    false   true   false
  | ComparerGt of x > y : comparer x y (x - y) (x - y)
     false   false   false   true    false  true
  | ComparerEq of x = y : comparer x y 0   0
     true    true    true    true    false  false.

Lemma ltrgtP x y : comparer x y `|x - y| `|y - x|
     (y == x) (x == y) (x <= y) (y <= x) (x < y) (x > y) .
```

Figure 4.7: Definition of `ltrgtP`

Example

```
R : realDomainType
x y : R
P : R -> R -> bool -> bool -> bool -> bool -> R -> Type
============================
P x y (x <= y) (y <= x) (x < y) (x == y) `|x - y|
```

```
case: (ltrgtP x y)
```

```
R : realDomainType
x y : R
P : R -> R -> bool -> bool -> bool -> bool -> R -> Type
============================
x < y -> P x y true false false false (y - x)

(* subgoal 2 is *)
y < x -> P x y false true true false (x - y)

(* subgoal 3 is *)
x = y -> P x y true false true true 0
```

Figure 4.8: Three way case analysis using an inductive family

4.3 Intervals

As soon as we start working with continuous functions (if only polynomials), intervals become pervasive objects in the statements we have to prove or the hypotheses present in the goal context. Intervals can be seen as sets defined by one or two linear order constraints, and interval membership as a conjunction of such constraints. Breaking down interval membership into such atomic constraints allows for the use of decision procedures for linear arithmetic to collect and solve the side conditions of interval membership. This approach however presents the unpleasant drawback of an explosion of the size of the context. Consider for instance the following trivial fact:

$$\forall a\ b\ c\ d\ x, \quad c \in [a,b] \wedge d \in [a,b] \wedge x \in [c,d] \Rightarrow x \in [a,b]$$

With the unbundled approach, proving this fact would lead to a Coq goal of the form we give in Figure 4.9.

Example

```
a b c d x : R
had : a <= d
hdb : d <= b
hac : a <= c
hcb : c <= b
hcx : c <= x
hxd : x <= d
========================
a <= x && x <= b
```

Figure 4.9: A non structured interval membership goal

Considering that on the way to prove a non trivial theorem, side conditions solved by this kind of easy facts are numerous and involve not only five but maybe much more values, this approach eventually requires the use of a decision procedure for linear arithmetic. A user may soon be overwhelmed by the number of constraints and unable to chain by hand the uninteresting steps of transitivity required to reach the desired inequality. One could argue this is not a serious problem since the decidability of this linear fragment and the implementation of the corresponding proof-producing decision procedures inside proof assistants is now folklore [10]. However, our experience is that the uncontrolled growth of the context and its lack of readability is an issue. We propose here a short infrastructure development which helps dealing with such interval conditions and helps improving the readability of the context by re-packing intervals and restoring the infix membership notation, with no extra effort from the user.

An interval bound is either a constant or infinity. We formalize interval bounds as a two cases inductive type parameterized by a type `T`, which does not need to be a real domain yet:

```
Inductive itv_bound (T : Type) : Type := BClose of bool & T | BInfty.
```

The constructor `BClose` builds a constant bound, which packages two pieces of information: the value of the bound, and a Boolean which indicates whether the extremity of the interval is open or closed. The constructor `BInfty` builds an infinite bound. Since the right or left position of the infinity symbol determines its interpretation as $+\infty$ or $-\infty$, this constructor does not need to package any extra information. Now, we formalize an interval as a pair of bounds, using an inductive type:

```
Inductive interval (T : Type) :=
  Interval of itv_bound T & itv_bound T.
```

with a single constructor `Interval` packaging two `(itv_bound T)` which are respectively the left bound and the right bound of the interval. We then define a bunch of notations `‘]a, b[`, `‘[a, b]`, `‘[a, +oo[` and all their variants with open or closed bounds as particular cases of these intervals. For example, the term:

```
Interval (BClose true a) (BClose false b)
```

is denoted by `‘[a, b[`. The second step of the infrastructure is to attach to any interval a predicate representing its actual characteristic function. For instance, the interval `‘[a, b[` is interpreted as `[pred x | a <= x < b]`, where `a`, `b` and `x` now need to be elements from a real domain `R`. At this stage, we can already rephrase the statement of our first example as the Coq goal in Figure 4.10.

Example

```
a b c d x : R
hd : d \in ‘[a, b]
hc : c \in ‘[a, b]
hx : x \in ‘[c, d]
========================
x \in ‘[a, b]
```

Figure 4.10: An interval membership goal.

The last step of our infrastructure is to provide generic tools to help the elementary proofs based on interval inclusion and membership. We start by converting a proof of interval membership into the list of constraints one can derive from this membership. We hence define a function `(itv_rewrite i x)` as in Figure 4.11, which performs a case analysis on its interval argument `i` and computes a conjunction of frequently used equalities that one can obtain as consequences of `(x \in i)`. The lemma `itvP` ensures that the interval membership `(x \in i)` implies this conjunction of rules. For instance, `(itv_rewrite ‘[a, b] x)` evaluates to the conjunction of: `(a <= x)`, `(x < a = false)`, `(x <= b)`, `(b < x = false)`, `(a <= b)` and `(b < a = false)`.

The enhanced version of the `rewrite` tactic we use [46] can take conjunctions (lists in fact) of rewriting rules as input: in that case, it rewrites with the first rule of the list which matches a sub-term of the current goal. Combined with the iteration switches of this same `rewrite` tactic, this feature helps creating on the fly a terminating rewrite system which can solve side conditions. The only purpose of the `itv_rewrite` function is to generate an appropriate rewrite system, by gathering a set of the constraints we can infer from an interval membership hypothesis.

```
Definition itv_rewrite (i : interval R) (x : R) : Prop :=
  let: Interval l u := i in
(match l with
   | BClose true a => (a <= x) * (x < a = false)
   | BClose false a => (a <= x) * (a < x) * (x <= a = false)
   | BInfty => forall x : R, x == x
 end *
match u with
   | BClose true b => (x <= b) * (b < x = false)
   | BClose false b => (x <= b) * (x < b) * (b <= x = false)
   | BInfty => forall x : R, x == x
 end *
match l, u with
   | BClose true a, BClose true b =>
     (a <= b) * (b < a = false) * (a \in `[a, b]) * (b \in `[a, b])
   | BClose true a, BClose false b =>
     (a <= b) * (a < b) * (b <= a = false) * (a \in `[a, b])
              * (a \in `[a, b[) * (b \in `[a, b]) * (b \in `]a, b])
   | BClose false a, BClose true b =>
     (a <= b) * (a < b) * (b <= a = false) * (a \in `[a, b])
              * (a \in `[a, b[) * (b \in `[a, b]) * (b \in `]a, b])
   | BClose false a, BClose false b =>
     (a <= b) * (a < b) * (b <= a = false) * (a \in `[a, b])
              * (a \in `[a, b[) * (b \in `[a, b]) * (b \in `]a, b])
   | _, _ => forall x : R, x == x
end)%type.


Lemma itvP (x : R) (i : interval R) : (x \in i) -> itv_rewrite i x.
```

Figure 4.11: Generating rewrite rules for intervals

We also provide tools to ease proofs of interval inclusion by programming a Boolean predicate `(subitv i1 i2)`, as defined in Figure 4.12, which reduces to an ordering constraint on the bounds of the intervals `i1` and `i2`. For instance the expression `(subitv '[c, d[ ']a, b[)` evaluates to `((a < c) && (d <= b))`.

The lemma `subitvP` from Figure 4.12, expresses that to show interval inclusion of `i1` into `i2`, it suffices that `(subitv i1 i2)` is true. We recall that, as presented in section 1.1, `{subset i1 <= i2}` is a notation for `(forall i, i \in i1 -> i \in i2)`.

Now our running example in Figure 4.10 can be solved using these facilities by the single line following command:

```
by apply: (subitvP _ hx); rewrite /= (itvP hc) (itvP hd).
```

The instantiation `(subitvP _ hx)` evaluates to this specialized statement of the theorem:

```
(subivt `[c, d] `[a, b]) -> {subset `[c, d] <= `[a, b]}
```

```
Definition le_boundl b1 b2 :=
  match b1, b2 with
    | BClose b1 x1, BClose b2 x2 => x1 < x2 ?<= if (b2 ==> b1)
    | BClose _ _, BInfty => false
    | _, _ => true
  end.

Definition le_boundr b1 b2 :=
  match b1, b2 with
    | BClose b1 x1, BClose b2 x2 => x1 < x2 ?<= if (b1 ==> b2)
    | BInfty, BClose _ _ => false
    | _, _ => true
  end.

Definition subitv (i1 i2 : interval R) : bool :=
  match i1, i2 with
    | Interval a1 b1, Interval a2 b2 => le_boundl a2 a1 && le_boundr
       b1 b2
  end.

Lemma subitvP : forall (i2 i1 : interval R),
 (subitv i1 i2) -> {subset i1 <= i2}.
```

Figure 4.12: Subinterval decision procedure

the application of which transforms the goal into `(subivt ‘[c, d] ‘[a, b])`. This goal in turn evaluates to `((a <= c) && (d <= b))` by computation thanks to the `/=` simplification switch. Finally, this latter goal is solved by rewriting the constraints related to the interval membership hypotheses on `c` and `d`.

This toolbox also contains facilities for interval splitting, in order to address the dichotomy processes commonly involved in root counting algorithms and proofs. We indeed provide a lemma `itv_splitU`

```
Lemma itv_splitU (xc : R) bc a b : xc \in Interval a b ->
  forall y, y \in Interval a b =
          (y \in Interval a (BClose bc xc))
       || (y \in Interval (BClose (~~bc) xc) b).
```

which partitions an interval $[a, b]$ into either the two disjoint intervals $[a, c[$ and $[c, b]$ or the two disjoint intervals $[a, c]$ and $]c, b]$, provided that c is in $[a, b]$.

For example, given elements `(a b c : R)` and a proof `c_ab` of the proposition `(c \in ‘[a b])`, the lemma `(itv_splitU true c_ab)` rewrites a term of the form `(y \in ‘[a, b])` into `(y \in ‘[a, c])|| (y \in ‘]c, b])`.

Comparison to existing interval theories

We can compare our work with Ioana Pasca's work on interval matrices in Coq [69], which aims at computing approximation errors on numbers. The latter formalization

is not a systematic study of interval arithmetic with open, closed or infinite bounds like ours, but a study specific of error intervals and computation on them.

The closest work to ours we could find is the one by Tobias Nipkow, Clemens Ballarin, Jeremy Avigad in Isabelle/HOL [64].

4.4 Structure of integers and rational numbers

We define data types for integers and rational numbers so that they are equipped with the appropriate real structures.

4.4.1 Integers

The source file for integers is `ssrint.v`. We define integers by joining two copies of natural numbers:

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

The `Posz` constructor is the natural injection of `nat` into `int`, but `(Negz n)` means $-(n+1)$, so that `(Negz 0)` is -1. Hence, each element of `int` represents a different integer. By declaring `Posz` as an implicit coercion, operations on integers can take natural arguments with no extra work.

We build stage by stage for `int` the decidable equality structure, choice structure, $\mathbb{Z}$-module structure, ring structure, commutative ring structure with decidable units, integral domain structure, numeric and real domain structures. We also show the canonical injection (`_%:R`) of `nat` into `int` coincides with `Posz`, and that it is a morphism for addition, multiplication, the order and norm.

We also define a special alternative to the numeric domain norm: the integer absolute value which has type `(int -> nat)`. For this operation, we use the same notation `` `|_| `` but within the `nat_scope` (delimited using `%N`), where the notation for the norm was in the `ring_scope` (delimited using `%R`). As expected, it satisfies the following property:

```
Lemma abszE (m : int) : `|m|%N = `|m|%R :> int.
```

4.4.2 Rational numbers

The source file for rational numbers is `rat.v`. We define rational numbers as a pair of an integer nominator and an integer denominator in a reduced form, which is such that the nominator and the denominator are coprime.

```
Record rat : Set := Rat {
  valq : int * int ;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}.
```

The operator `coprime` has type `(nat -> nat -> nat)`: it acts on natural numbers. In order to apply it to integers, we first cast them to natural numbers using the integer absolute value defined in Section 4.4.1.

We build stage by stage for `rat` the decidable equality structure, choice structure, $\mathbb{Z}$-module structure, ring structure, commutative ring structure with decidable units, integral domain structure, field structure, numeric and real domain structures,

numeric and field structures and Archimedean field structure. We provide tools to write a rational number as the division of an integer by a positive natural, such that these two numbers are coprime. We also build the morphism `ratr : rat -> R` mapping rationals to any ring with decidable units and we show this is the only morphism from `rat` to `R`.

5 Cauchy reals, algebraics

This chapter is devoted to the construction of algebraic numbers, seen as a restriction of constructive real numbers. We first describe a construction of real numbers modeled by Cauchy sequences and we build the operations on those real numbers. We then take the subtype of reals that are roots of some monic polynomial (*i.e.* with a leading coefficient equal to 1) and transfer all the arithmetic of real numbers on this restriction, which involves polynomial arithmetic.

From now on, we denote by F an Archimedean field equipped with a decidable equality structure and with a choice structure. All the constructions are done over F, but one can think of F as being the Archimedean field of rational numbers $\mathbb{Q}$. Although it is necessary for this construction, we do not detail the use of the Archimedean property for the sake of readability.

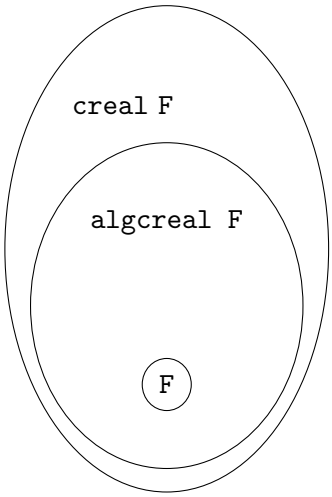


Figure 5.1: The construction of algebraic Cauchy reals

5.1 Cauchy reals

5.1.1 Cauchy sequences

We define a Cauchy sequence x as a sequence $(x_n)_{n\in\mathbb{N}}$ in $F^{\mathbb{N}}$, together with a convergence modulus $m_x : F \to \mathbb{N}$ such that from the index $m_x(\varepsilon)$, the distance between any two elements is smaller than ε. We call this property the *Cauchy property* and it can be expressed by the formula

$$\forall \varepsilon\ \forall i\ \forall j\ ,\ m_x(\varepsilon) \leq i\ \wedge\ m_x(\varepsilon) \leq j\ \Rightarrow\ |x_i - x_j| < \varepsilon.$$

We will denote by x both the sequence and the sequence together with its convergence modulus and Cauchy property. We represent sequences of elements of F as functions from natural numbers to F. Hence, we formalize Cauchy reals by packaging together the representation of a sequence $(x_n)_n$, a modulus m_x and the *Cauchy property*:

```
Definition creal_axiom (x : nat -> F) :=
  { m : F -> nat
  | forall ε i j, m ε <= i -> m ε <= j -> `|x i - x j| < ε}.
Inductive creal := CReal
  {cauchyseq :> nat -> F;
           _ :  creal_axiom cauchyseq}.
```

Technical remark

We draw the reader attention to the fact that `cauchyseq` is declared as an implicit coercion from `creal` to `(nat -> F)`, so that any Cauchy real `x` can be used like a sequence through the automatic insertion of the projection `cauchyseq` if needed. Hence the confusion between Cauchy sequences and simple sequences can be made in Coq as well.

We also remind that `{m : F -> nat | ...}` denotes a Σ-type and can be read "there exist a function `(m : F -> nat)` such that ...". In addition, this convergence modulus m_x of a Cauchy real x is then formalized as a function `(cauchymod x)` of type `(F -> nat)`.

Technical remark

A Coq user will remark that such a function is definable because the existential modulus in the definition of the Cauchy sequence is in `Type`. By definition of Cauchy sequences, it satisfies the following property:

```
Lemma cauchymodP (x : creal) (ε : F) (i j : nat) :
  cauchymod x ε <= i -> cauchymod x ε <= j -> `|x i - x j| < ε
```

It is important to note that when we apply this lemma, we produce a sub-goal, which we call side condition, of the form $f(\varepsilon) \leq i$, where f is `(cauchymod x)` in the latter instance. This is a general scheme in our development: during a proof we may generate a finite number n of side conditions $f_k(\varepsilon) \leq i$ for $k \in \{1, \ldots, n\}$. Indeed, if all constraints on i are formulated like this, it suffices to take i to be the maximum of all the $f_k(\varepsilon)$, in order to satisfy all the side conditions on i. We have designed, as detailed in Section 5.1.4, an automated procedure to solve this kind of constraints

using the `Ltac` language [33] available in Coq, so that many proofs begin with a command meaning "let i be a big enough natural number".

From `cauchymod` we can define a function `ubound` to bound above the values of elements of a Cauchy sequence. It then satisfies the following property:

```
Lemma uboundP : forall (x : creal) (i : nat), `|x i| <= ubound x.
```

In the rest of the development, this function is used to compute the convergence moduli of numerous Cauchy sequences. We use the notation $\lceil x \rceil$ for the upper bound `(ubound x)`. The lower bound $\lfloor x \rfloor$ is a positive number which bounds bellow the sequence $(|x_n|)_n$. It cannot yet be defined since it exists only if the real is not zero: this is the problematic of Section 5.1.2.

5.1.2 Equality on Cauchy reals

Although Cauchy reals are represented by sequences, which are in turn formalized as functions, the notion of equality on Cauchy reals is not intentional equality on function nor point-wise equality on functions, but equality up to convergence. Indeed, two Cauchy sequences x and y may have no index n such that $x_n = y_n$, but could still represent the same real number as long as $(x_n)_n$ and $(y_n)_n$ converge to the same real number, in a classical sense. The equivalence defined by $x \equiv y$ if and only if $\lim_{n\to\infty}(x_n - y_n) = 0$, identifies Cauchy sequences up to convergence, and hence models the equality on Cauchy reals.

The Coq system provides a mechanism called *setoid* [4]. A setoid is a type `T` with an equivalence relation `R` on it. We call *setoid equality* this equivalence relation. Operations on a setoid are defined on its base type `T`, with the constraint that they must be compatible with the setoid equality. An operation is compatible with the setoid equality if given two sets of arguments which are pairwise identified by the setoid equality, the operation computes values that are identified by the setoid equality:

```
forall (x x' y y' : T), R x x' -> R y y' -> R (op x y) (op x' y').
```

Moreover, Coq provides tools to declare setoids and functions that are compatible with the setoid equality. It extends the `rewrite` tactic in order it to work not only with standard equality `(_ = _)` but also with the setoid equality `R` of any declared setoid. Of course, this extension of `rewrite` changes a term `(P x)` into a term `(P y)` when `(R x y)` and only if the context `P` is compatible with the setoid equality `R`.

Given two Cauchy reals x and y, if there exists a quantity δ and a rank k such that $\delta \leq |x_i - y_i|$ for all i greater than k, then the two sequences cannot converge to the same value. Conversely, if two Cauchy sequences do not converge to the same value, *i.e.* $\lim_{n\to\infty} x_n - y_n \neq 0$, then there must exist a δ and a rank k such that $\delta \leq |x_i - y_i|$ for all i greater than k, which means that a program could find the δ. Thus, the comparison of Cauchy reals is semi-decidable. However it is not decidable (see [13] for example).

The primitive notion for comparison is not equality but apartness, which contains additional information: a witness for the non-negative lower bound of the gap separating the two sequences. We now write $x \neq y$ for apartness and $x \equiv y$ for its negation. The notion of non apartness coincides with the notion of equivalence stated above and is declared as the setoid equality on Cauchy reals. From a proof of apartness $x \neq y$ we must be able to extract a rank k and a positive witness δ

which bounds below the sequence $(|x_n - y_n|)_n$ from the rank k. This lower bound is needed to define the inverse as described in Section 5.1.3.

We define apartness as follows, and we pose the notation `(_ != _)` on this definition:

```
Definition neq_creal (x y : creal) : Prop :=
  exists δ, (0 < δ)
         && (3 * δ <= `|x (cauchymod x δ) - y (cauchymod y δ)|).
Notation "x != y" := (neq_creal x y).
Notation "x == y" := (~ (x != y)).
```

And now, we need to extract a lower bound for $(|x_n - y_n|)_n$, from of proof of $x \neq y$. In order to do that, we use the `xchoose` operator seen in Section 2.2 to extract a positive witness δ as a function from the apartness predicate to F:

```
Definition lbound x y (neq_xy : x != y) : F := xchoose F _ neq_xy.
```

Now, let us prove that this δ is indeed the lower bound we are looking for. Given two Cauchy reals x and y which are provably apart from each other, let δ be their positive witness of separation as defined above. From `xchooseP` we get that $3\delta \leq \left|x_{m_x(\delta)} - y_{m_y(\delta)}\right|$. Thus using the triangular inequality, we get

$$\forall i, \quad 3\delta \leq \left|x_{m_x(\delta)} - x_i\right| + |x_i - y_i| + \left|y_i - y_{m_y(\delta)}\right|.$$

Thanks to the Cauchy property, we know that $\left|x_{m_x(\delta)} - x_i\right| < \delta$ as soon as i is bigger than $m_x(\delta)$ and that $\left|y_i - y_{m_y(\delta)}\right| < \delta$ when i is bigger than $m_x(\delta)$. Putting this together, if i is bigger than $\max(m_x(\delta), m_y(\delta))$ we get that

$$\forall\, i \geq \max(m_x(\delta), m_y(\delta)), \quad \delta \leq |x_i - y_i|.$$

Now, by formalizing this reasoning in Coq, we get the lemma

```
Lemma lboundP (x y : creal) (neq_xy : x != y) (i : nat) :
  cauchymod x (lbound neq_xy) <= i ->
  cauchymod y (lbound neq_xy) <= i -> lbound neq_xy <= `|x i - y i|.
```

Which means that the `lbound` function is indeed a lower bound of separation of any two elements x_i and y_i of the sequences, for big enough indexes i.

From this, we also derive a particular case:

```
Lemma lbound0P (x : creal) (x_neq0 : x != 0) (i : nat) :
  cauchymod x (lbound x_neq0) <= i ->
  cauchymod 0 (lbound x_neq0) <= i -> lbound x_neq0 <= `|x i|.
```



Technical remark

We could have defined apartness as follows:

```
Definition apart_creal x y : Type := {δ : F | 0 < δ &
  forall i, cauchymod x δ <= i ->
            cauchymod y δ <= i -> δ <= `|x i - y i|}.
```

This definition `apart_creal` is using a witness in `Type` to make it directly available. In order to be fully compatible with the setoid mechanism, the apartness must be in `Prop`, not in `Type`. But if it were in `Prop` we could not make a function that extract the witness, nor could we use `xchoose` to turn the existential in `Prop` into an existential in `Type`. Indeed the predicate on which the existential applies is not decidable.

We also build a function `eq_mod` which takes a proof equality of two cauchy reals `x` and `y` and a positive number ε, and returns a rank from which the distance between `x` and `y` is upper bounded by ε.

```
Definition eq_mod (x y : creal) : (x == y) -> F -> nat.
Lemma eq_modP (x y : creal) (eq_xy : x == y) (ε : F) (i : nat) :
   0 < ε -> (eq_mod eq_xy ε <= i) -> `|x i - y i| < ε.
```

Order relation

The order relation is handled the same way as apartness. The primitive notion is the strict ordering, the negation of which defines the non-strict ordering.

```
Definition lt_creal (x y : creal) : Prop := exists ε, (0 < ε) &&
    (x (cauchymod x ε) + ε * 3 <= y (cauchymod y ε)).
Notation "x < y" := (lt_creal x y) : creal_scope.
Notation "x <= y" := (~ (y < x)).
```

A remarkable fact is that ordering is derivable from a proof of apartness:

```
Lemma neq_ltVgt (x y : creal) : x != y -> {x < y} + {y < x}.
```

where the operator `(_ + _)` is the disjunction in `Type`.

We also build `diff` and `le_mod` to be the equivalent of respectively `lbound` and `eq_mod`, which satisfy the following specifications:

```
Lemma diffP (x y : creal) (lt_xy : (x < y)) i :
  cauchymod x (diff lt_xy) <= i ->
  cauchymod y (diff lt_xy) <= i -> x i + diff lt_xy <= y i.

Lemma le_modP (x y : creal) (le_xy : (x <= y)) ε i :
  0 < ε -> le_mod le_xy ε <= i -> x i < y i + ε.
```

5.1.3 Arithmetic operations and bounding

We build negation, addition and multiplication on Cauchy reals and prove their output are Cauchy sequences in a systematic way: we perform the appropriate operation on each element of the sequence and we forge a convergence modulus for each operation.

Negation, addition and multiplication. We exhibit the convergence moduli of negation, addition and multiplication of Cauchy reals. Given the convergence moduli m_x of x and m_y of y, we prove the convergence moduli of $(-x_n)_n$, $(x_n+y_n)_n$ and $(x_n y_n)_n$ are respectively: $m_x, \varepsilon \mapsto \max\left(m_x\left(\frac{\varepsilon}{2}\right), m_y\left(\frac{\varepsilon}{2}\right)\right)$ and $\varepsilon \mapsto \max\left(m_x\left(\frac{\varepsilon}{2\lceil|y|\rceil}\right), m_y\left(\frac{\varepsilon}{2\lceil|x|\rceil}\right)\right)$.

Inverse. We need to know a non-negative lower bound δ for the sequence $(|x_n|)_n$ of absolute values from some arbitrary rank, and use it to prove that the sequence of point-wise inverses $(\frac{1}{x_n})_n$ is a Cauchy sequence. According to Section 5.1.2, such a positive lower bound δ is given by `(lbound x_neq0)` when given a proof `(x_neq0 : x != 0)` that x is apart from 0 (in the sense of Cauchy sequences). This value δ is such that $\forall i > m_x(\delta),\ \delta \leq |x_i|$.

If i and j are greater than $m_x(\varepsilon\delta^2)$, we have $|x_i - x_j| < \varepsilon\delta^2$. By definition of δ and if i and j are greater than $m_x(\delta)$, we get $\delta \leq |x_i|$ and $\delta \leq |x_j|$, thus $|x_i - x_j| < \varepsilon|x_i x_j|$. And finally:

$$\left|\frac{1}{x_i} - \frac{1}{x_j}\right| < \varepsilon$$

Thus, a convergence modulus is $\varepsilon \mapsto \max\left(m_x(\varepsilon\delta^2), m_x(\delta)\right)$.

Morphism property of arithmetic operations. We can check that all arithmetic operations are compatible with the equality for Cauchy sequences, using a simple point-wise study. The order relation is also compatible. However, there is no need to systematically study the compatibility with apartness.

Bounds and evaluation for polynomials

We now introduce several bounds on polynomials which are useful when doing analysis. We define them using the Taylor expansion of polynomial P.

$$\begin{aligned}
B_0(P,c,r) &= 1 + \sum_{i=0}^{n} |p_i|(|c|+|r|)^i \\
B_1(P,c,r) &= \max(1,2r)^n \left(1 + \sum_{i=1}^{n} \frac{B_0(P^{(i)},c,r)}{i!}\right) \\
B_2(P,c,r) &= \max(1,2r)^{n-1} \left(1 + \sum_{i=2}^{n} \frac{B_0(P^{(i)},c,r)}{i!}\right)
\end{aligned}$$

where $P^{(i)}$ is the i^{th} derivative of the polynomial P. These bounds satisfy the following properties, for all x and y in $[c-r, c+r]$:

$$\begin{aligned}
|P(x)| &\leq B_0(P,c,r) \\
|P(y) - P(x)| &\leq |y-x| B_1(P,c,r) \\
\left|\frac{P(y)-P(x)}{y-x} - P'(x)\right| &\leq |y-x| B_2(P,c,r)
\end{aligned}$$

These bounds are constructive witnesses for well-known classical mathematical results on continuous and derivable functions, specialized to univariate polynomials. The bound B_0 is only an intermediate step to bounds B_1 and B_2. The bound B_2

is used in Section 7.1.2 to prove that polynomials whose derivative does not change sign on an interval are monotone on this interval.

The bound B_1 is used to show that polynomial evaluation preserves the Cauchy property for sequences. Indeed, we build polynomial evaluation of a polynomial $P \in F[X]$ in a Cauchy real as the point-wise operation, and in order to prove that the result is a Cauchy sequence, we bound $|P(x) - P(y)|$ when $|x - y|$ is small enough. The convergence modulus is given by $\varepsilon \mapsto m_x\left(\frac{\varepsilon}{B_1(P,0,\lceil x \rceil)}\right)$. We then prove that $P(x) \neq P(y) \Rightarrow x \neq y$, which implies that $x \equiv y \Rightarrow P(x) \equiv P(y)$. Hence the evaluation of a polynomial at a Cauchy real is compatible with the equality of Cauchy reals.

5.1.4 Reasoning with big enough values

In analysis, we use the idiom "for big enough values of i ...". Using this presentation, mathematicians do not always write on the paper the values that this "big" i could take.

To illustrate this, we show two examples. First, a very basic fact about Cauchy reals, which we claimed true in the end of Section 5.1.3.

Example

Lemma 5.1 (multiplication of Cauchy reals is a morphism for $\equiv$)**.** *Given four Cauchy reals x, x' and y, y', if $x \equiv x'$ and $y \equiv y'$ then $xy \equiv x'y'$.*

Proof. It suffices to show that for all positive ε in F, there exists i, such that if n is bigger than i, we have $|x_n y_n - x'_n y'_n| < \varepsilon$.
So let ε be a positive number in F and let i be a big enough natural number and n bigger than i. It suffices to show that

$$\begin{cases} |x_n - x'_n||y_n| < \frac{e}{2} \\ |x'_n||y_n - y'_n| < \frac{e}{2}. \end{cases}$$

Which is equivalent to

$$\begin{cases} |x_n - x'_n| < \frac{e}{2\lceil y \rceil} \\ |y_n - y'_n| < \frac{e}{2\lceil x' \rceil}. \end{cases}$$

Now, since $x \equiv x'$ and $y \equiv y'$, by taking i big enough, we can satisfy those two conditions.

□

This second example is a key lemma in the development of algebraic Cauchy reals, which we use in the proof of decidability of the comparison in Section 5.2.1.

Example

Lemma 5.2. *Given two coprime polynomials P and Q with coefficients in F, and given a Cauchy real x such that $PQ(x)$ is zero (i.e. is equivalent to zero, as a Cauchy real), we can decide which of P or Q is the one which annihilates x.*

Example (continued)

Proof. First, since P is coprime with Q, using Bézout lemma we get two polynomials U and V in $F[X]$ such that $UP+VQ=1$. Let i be a natural number.

If $(UP)(x_i) > \frac{1}{2}$**,** then let us show $Q(x)$ is zero. Let us suppose that $Q(x)$ is nonzero and find a contradiction (this is constructive since equivalence is non apartness). By hypothesis, $PQ(x)$ is zero so it suffices to show that $P(x)$ is nonzero to find a contradiction (it is very easy to show that if two Cauchy reals are nonzero, then their product also is). So it suffices to show that $(UP)(x)$ is nonzero. But if i was chosen big enough, then for all n bigger than i, $(UP)(x_n)$ is close enough to $(UP)(x_i)$, which is greater than $\frac{1}{2}$. Thus $((UP)(x_n))_n$ cannot converge to zero, which means $(UP)(x)$ is not equal to zero (as a Cauchy real).

If $(UP)(x_i) \leq \frac{1}{2}$**,** let us show that $P(x)$ is zero. The beginning is the same: it suffices to show that $VQ(x)$ is not zero. But since $(UP)(x_i) + VQ(x_i) = 1$ and $(UP)(x_i) \leq \frac{1}{2}$, we get that $(VQ)(x_i) \geq \frac{1}{2}$. But if i is big enough, for all n bigger than i, $(VQ)(x_n)$ is close enough to $(VQ)(x_i)$, which is greater than $\frac{1}{2}$. Thus $((VQ)(x_n))_n$ cannot converge to zero, which means $(VQ)(x)$ is nonzero. □

When written as such, it is not obvious to a human reader that i can be chosen to be big enough for both cases simultaneously. Moreover, it is not rigorous to introduce a variable (implicitly quantifier universally) and impose some conditions on it on the middle of a proof. However, we managed to make Coq accept the proof in this exact proof style, hence providing a formal proof using a seemingly fuzzy reasoning.

An very common error, which is difficult to detect, is that in some case we may want i to be bigger than a quantity which already depends on i, which is can be unsound. To be rigorous, one has to say beforehand how big i should be, thus writing by hand arbitrarily complicated lower bounds on i before beginning the proof. However, the proof is usually independent from the actual value of i and relies only on its existence.

Of course, Coq must be provided the lower bounds for i anyway, and the user could manually enter a values himself. The success in entering values guarantees the non circularity of dependencies in i, and the success in completing the proof with them shows that i is big enough indeed. A very important remark is that we can often make these bounds appear naturally during the proof, by stating theorems about asymptotic facts in a special way. In fact, in Section 5.1.1 we have carefully stated lemmas in the appropriate way. Indeed, `cauchymodP`, `lboundP`, `eq_modP`, `diffP` and `le_modP` all have the same form: their hypotheses are all of the form $f(\varepsilon) \leq i$ for some f. This means that when we apply (or rewrite with) one of these lemma, we generate a sub-goal of the form $f(\varepsilon) \leq i$ for some f. The only constraints that appear on the i involved in those expressions are that it must be bigger than a finite number of expressions of the form $f(\varepsilon)$. Thus by taking i to be the maximum of all those value, i satisfies these preconditions. We call such a i a *big enough* natural number.

We designed a Coq tactic that creates a big enough number by setting it as the

maximum of an unknown list (existential variables). And we also created a tactic which solves any goal of the form $f(\varepsilon) \leq i$ by adding $f(\varepsilon)$ to the list. With this methodology, it is not necessary to provide lower bound for i beforehand, because we can progressively extend the list of numbers i should be bigger than. In other words, our tactics helps the user to reason like in paper mathematics by letting him pose an arbitrary big value and explain later why it can be big enough. This can be seen as a form of proof irrelevance, but for some proper data. Indeed the i is a number which is not relevant and which we make opaque so that other functions and lemmas cannot rely on its value. Using these, we completed the two proofs given in example using the same proof style we used on the paper.

The strong points of these tactics is that they hide the use of existential variables, which can be dangerous to let a final user handle. They handle well the case where the lower bound $f(\varepsilon)$ is a local definition, as long as it does not depend from the big enough number. Moreover, these tactics work on integers, but it might be interesting to generalize it to "small enough" element, even though in an Archimedian domain it could be simulated by $\frac{1}{i}$ for a big enough i. We could also generalize this for any lattice or for classes of predicates which cannot get empty by intersecting them with each other a finite number of time.

We believe this methodology has applications in many developments on the formalization of mathematics, in analysis and in algebra. We already started to use it (successfully) for a construction of multivariate polynomials (polynomials with an arbitrary number of indeterminates). And we believe that by extending big enough to thin enough intervals, it could ease the management of neighborhoods in Section 9.3. Moreover we believe a wide range of problems in analysis can benefit from this: the formalization of limits, equivalents, delimited developments of univariate and multivariate functions.



Technical remark

We implemented the tactic in `Ltac`. It relies on SSReflect pattern selection mechanism [47] to ensure the robustness of our tactic: we select sequentially all subterms of the form $f(\varepsilon) < n$, and try to apply our tactic to each of them. There is still work to do to improve error messages and make these tactics more ergonomic.

For example, for now, the user has to provide the empty sequence by hand to close the existential variable when he has finished using his big enough number $i = \max[f_1(\varepsilon), f_2(\varepsilon), \ldots, ?]$. Before Coq version 8.4, we even had to create artificially a fake goal, just to provide the opportunity to the user to close the sequence using the empty sequence. From Coq version 8.4, the system provides a command `Grab Existential Variables` which make this trick less useful but the user still has to input `nil` by hand, requiring a line of script saying nothing useful. A solution we find ergonomic would be to provide a potential default value at the creation of some existential variables and use it only if necessary at `Qed` time.

Another refinement we have developed (only for v8.4 so far) overloads the SSReflect `//` switch. This switch is used to discharge automatically all the trivial subgoals a `rewrite` can generate, and we believe sub-goals of the form $f(\varepsilon) < i$ belong to this category.

5.1.5 Comparison with other implementations of reals

It remains unclear whether an axiomatization of Cauchy reals as described in [39] would fit our needs. At some point of the development, given a Cauchy real x, we need to be able to find an approximation in `F` of x with an arbitrary precision ε. And it seems that the latter axiomatization does not capture approximation in the base field.

The C-CoRN library also provides an interface for Cauchy reals in [39] and a construction of Cauchy reals by Russell O'Connor's [66], which is used to instantiate the interface. Although their definition is close enough to ours, we redefine and re-implement Cauchy reals from scratch, mainly because our algebraic structures are incompatible and because we depend on an arbitrary discrete Archimedean base field F. We use this as an opportunity to restate the definitions in a way which is more compatible with our proof style. Moreover our implementation is shorter and more direct, but less efficient, when compared with Russell O'Connor's [66].

Robbert Krebbers and Bas Spitters [59] already encountered in C-CoRN the problem which led us to define `neq_creal` instead of `apart_creal` back in Section 5.1.2. Like us they did chose a definition in `Prop` and they extracted the witness with some effort, using the "constructive indefinite description" theorem, which is provable for decidable properties whose domain is `nat`. In fact, our solution uses a variant of this theorem — the `xchoose` function defined in Section 2.2 — to get the witness almost directly.

5.2 Algebraic reals

Now, we formalize real algebraic numbers on top of Cauchy reals.

```
Inductive algcreal := AlgCReal {
  creal_of_alg : creal;
  annul_algcreal : {poly F};
  _ : monic annul_algcreal;
  _ : annul_algcreal.[creal_of_alg] == 0
}.
```

Here, an algebraic Cauchy real `(AlgCReal x P monic_P root_P_x)` represents an algebraic number as a Cauchy real `x` and a polynomial `P` with a proof `monic_P` that `P` is monic and a proof `root_P_x` that `x` is a root of `P`. We recall that he notation `p.[x]` stands for polynomial evaluation in the source code. In mathematical notations, we will represent algebraic Cauchy reals by pairs (x, P) of a Cauchy real an a polynomial in $F[X]$ which annihilates it (*i.e.* $P(x)$ is zero).

First we prove that Cauchy reals setoid equality is decidable on algebraic Cauchy reals. Then we build arithmetic operations.

5.2.1 Decidability of comparison

Whereas the comparison on Cauchy reals is only semi-decidable, the comparison on algebraic Cauchy reals is decidable. We call `eq_algcreal` this decision procedure. It uses the additional data given by the annihilating polynomials. In fact, we only need to decide if some algebraic Cauchy real is zero, because we can test whether $x = y$

by comparing $x - y$ to zero. Hence, we have to define subtraction first, but for the sake of readability, we only define it in Section 5.2.2.

Before explaining how to compare a Cauchy real to zero, we have to explain a few prerequisites.

Theorem 5.3. *Given two polynomials P and Q with coefficients in F, and given a Cauchy real x such that $PQ(x)$ is zero, we can decide which of P or Q is the one which annihilates x.*

This theorem is stronger than Lemma 5.2 because it does not need the polynomials to be coprime. The idea of the proof is to decompose P and Q into factors that are coprime with each other and to apply Lemma 5.2 to them. Thanks to this theorem we can prove the following lemma:

Lemma 5.4. *Let x be a Cauchy real and P be a monic polynomial annihilating x. If there exists a polynomial Q that does not divide P but which is not coprime with Q, then there exists a monic polynomial R of degree smaller than P, such that R also annihilates x.*

This lemma expresses that given an algebraic Cauchy real (x, P) we can find an equivalent algebraic Cauchy real (x, R) where the degree of R is smaller than the degree of P, provided that there exists a polynomial Q which shares a factor with P without being a multiple of P.

Now, let (x, P) be an algebraic Cauchy real we wish to compare to zero, so P is the monic annihilating polynomial of the Cauchy real x. We reason by induction on the degree of P. The base case is trivial since P cannot be zero, because it is monic. For the induction step, there are two possibilities.

Either the indeterminate X is coprime with P, then zero is not a root of P, thus $x \neq 0$.

Or X and P are not coprime, in which case, if $P = X$ then $x \equiv 0$. So let us suppose P is not X, and since they are not monic and not coprime, it means that X is a proper divisor of P, hence P do not divide X. By Lemma 5.4, there exists a polynomial R of degree strictly smaller than P, such that (x, R) represents the same algebraic Cauchy real. We apply the induction hypothesis to (x, R), and we are done.

5.2.2 Arithmetic operations

We build arithmetic operations (negation, addition, multiplication, inverse) from the constants 0 and 1 and using the subtraction and the division. The embedding of the constants $c \in F$ is obtained from the pair $(c, X - c)$ (here c is a constant Cauchy sequence).

In the remainder of this section we consider two algebraic Cauchy reals x and y, whose respective Cauchy sequences are x and y, and whose respective annihilating monic polynomials are P and Q.

Let us recall (Section 5.1.3) that the subtraction $x-y$ (resp. division $\frac{x}{y}$) is obtained as the point-wise subtraction (resp. division) of elements of the sequence. Let us find a polynomial whose root is this new sequence.

Subtraction

Our candidate is the following resultant (see Section 3.2 for the definition of resultants bi-variate polynomials):

$$R(Y) = \mathrm{Res}_X\left(P(X+Y), Q(X)\right).$$

There are two essential properties to prove about this resultant: it is nonzero and it annihilates the subtraction.

R is nonzero. Let us suppose that R is zero and find a contradiction. Since R is zero, $P(X+Y)$ and $Q(X)$ are not coprime.

Thanks to the corollary to Bézout theorem described in Section 3.1, and applied to polynomials in the indeterminate Y with coefficients in $F[X]$ we know there exist $U, V \in F[X,Y]$ such that U and V are nonzero, $\deg_X(U) < \deg(Q)$ and

$$U(X,Y)P(X+Y) = V(X,Y)Q(X).$$

Let the polynomials u and v in $F[X]$ be the respective Y-leading coefficients of U and V. Let p in F be the leading coefficient of P. By taking the Y-leading coefficient in the previous formula, we get

$$u(X)p = v(X)Q(X).$$

This equation gives that $\deg(Q) \leq \deg(u)$, but $\deg(u) \leq \deg_X(U) < \deg(Q)$. This is a contradiction.

R annihilates the subtraction. Let us prove that R annihilates the Cauchy sequence $x - y$. Since R is in the ideal generated by $P(X+Y)$ and $Q(X)$, there exist U and V such that

$$R(Y) = U(X,Y)P(X+Y) + V(X,Y)Q(X).$$

Hence, for all natural number n, by evaluation at $X = y_n$ and $Y = (x_n - y_n)$, we get

$$R(x_n - y_n) = U(y_n, x_n - y_n)P(x_n) + V(y_n, x_n - y_n)Q(y_n).$$

But $P(x) \equiv 0$ and $Q(y) \equiv 0$. As x_n and y_n are bounded (respectively by $\lceil x \rceil$ and $\lceil y \rceil$) and U is bounded on a bounded domain (cf Section 5.1.3) we have that $R(x-y) \equiv 0$.

Remark that now the subtraction is defined, we can decide the equality of two arbitrary values by comparing their subtraction to zero, using the result from Section 5.2.1.

Division

When the algebraic Cauchy real y is zero, an annihilating polynomial is X. When y is nonzero, we can find a new Q annihilating y such that $Q(0) \neq 0$. The resultant

$$R(Y) = \mathrm{Res}_X\left(P(XY), Q(X)\right)$$

is a candidate for being an annihilating polynomial of $\frac{x}{y}$.

R is nonzero. Let us suppose that R is zero and let us find a contradiction. Since R is zero, $P(XY)$ and $Q(X)$ are not coprime.

Thanks to the corollary to Bézout theorem described in Section 3.1, we know there exist $U, V \in F[X, Y]$ such that U and V are nonzero, $\deg_X(U) < \deg(Q)$ and

$$U(X, Y)P(XY) = V(X, Y)Q(X).$$

By evaluation at $Y = 0$ we get:

$$U(X, 0)P(0) = V(X, 0)Q(X).$$

We reason by induction on the degree of V. The base case is trivial as V cannot be null.

If no member cancels, this equation gives $\deg(Q) \leq \deg(U(X, 0))$. But we also had $\deg(U(X, 0)) \leq \deg_X(U) < \deg(Q)$, which brings a contradiction.

Let us suppose that one of the member cancels, so both of them do. This means that both $V(X, 0)Q(X)$ and $U(X, 0)P(0)$ are zero. But as $F[Y]$ is an integral domain and Q is nonzero (by hypothesis), we get that $V(X, 0)$ is zero and that either $U(X, 0) = 0$ or $P(0) = 0$. Since $V(X, 0) = 0$ we know that $Y|V(X, Y)$. Now let us discuss on the two possibilities for the first member.

- If $U(X, 0)$ is zero, then $Y|U(X, Y)$. Hence, by dividing both members of the equation by Y, there exists $U'(X, Y)$ and $V'(X, Y)$, whose degrees in Y are strictly smaller than the ones of U and V, and such that

  $$U'(X, Y)P(XY) = V'(X, Y)Q(X).$$

- If $P(0)$ is zero, then $X|P(X)$. This implies that $XY|P(XY)$. But also know that

  $$U(0, Y)P(0) = V(0, Y)Q(0).$$

  And since $Q(0) \neq 0$, we necessarily have $V(0, Y) = 0$. It follows that $X|V(X, Y)$ and as we knew that $Y|V(X, Y)$, we find that $XY|V(X, Y)$. Thus, there exist P' and V' whose degrees are strictly smaller than those of P and V respectively, such that

  $$U(X, Y)P'(XY) = V'(X, Y)Q(X).$$

In both cases, we get an equation of the previous form, where V has a smaller degree.

R annihilates the division. In the same way we did for subtraction, we show that $R(\frac{x}{y}) \equiv 0$.

6 Quotient types in Coq

Given some base set S and an equivalence $\equiv$, one may see the quotient $(S \,/\, \equiv)$ as the partition $\{\pi(x) \mid x \in S\}$ of S into the sets $\pi(x) \mathrel{\hat{=}} \{y \in S \mid x \equiv y\}$, which are called the equivalence classes of x.

In this chapter, we focus on the specification, the theory and the construction of quotient types themselves, as explained in Section 6.1. We first describe our definition of a quotient interface in Section 6.2. We show in which way it captures the desired properties of quotients and how we instrumented type inference to help the user to be concise. Surprisingly, this interface does not rely on an equivalence relation in its axiomatization, so we explain how we recover quotients by an equivalence relation from it. We conclude in Section 6.4 by comparing our approach to related work.

6.1 Quotients in mathematics

We call base set (respectively setoid and type) the set (respectively setoid and type) we quotient, and the result is called the quotient set (respectively quotient setoid and quotient type).

We distinguish several uses of quotients in the literature. On the one hand, we have structuring quotients, where the quotient can often be equipped with more structure than the base set. For instance, the quotient of pairs of integer to get rational numbers can be equipped with a structure of field. Similarly, the quotient of the free algebra of terms generated by constants, variables, sum and products gives multivariate polynomials (*i.e.* polynomials with arbitrarily many variables). This kind of quotient is often left implicit in mathematical papers.

On the other hand, we have algebraic quotients, for which we can transfer the structure from the base set to the quotient. For instance, the quotient of a group by a normal subgroup or the quotient of a ring by an ideal belong to this category. For those quotients, the structure on the base set and on the quotient set matters and the canonical surjection onto the quotient is a morphism for this structure.

In type theory, there are two known options to represent the notion of quotient. The first option is to consider quotients of setoids. A setoid is a type with an equivalence relation called setoid equality (as we already described and used in Section 5.1.2). Now, quotienting a setoid (the base setoid) amounts to changing the

setoid equality to a broader one. However, we still consider elements from the base type, *i.e.* the type underlying both the base setoid and the quotient setoid. This point of view is more the study of equivalence relation than the study of quotients. Moreover, although rewriting with setoid equality is supported by the system [79], it is still not as practical nor efficient as rewriting with Leibniz equality.

The second option is to forge a quotient type, *i.e.* a type where each elements represent one and only one equivalence class of the base type. This point of view leads to study the quotient as a new type, on which equality is the standard (Leibniz) equality of the system. In this framework, the equivalence that led to the quotient type is not a primitive notion.

6.2 An interface for quotients

We define a quotient of a base type `T`, to be any type `qT` which can be characterized by a canonical surjection function `pi` which associates to elements of the type `T` their corresponding element in `qT`, and a representative function `repr` which selects a unique representant in `T` for each element in `qT`. The composition of the representative function with the canonical function must be the identity of the quotient type: the representant of any class must be sent to the initial class by the canonical surjection. This is the axiom of quotient types, and we call it `reprK`[1] and it states that `forall x, pi (repr x)= x` (see Figure 6.2).

A small interface

The interface for quotients must have a field for the quotient type, a field for the representative function `repr`, a field for the canonical surjection `pi` and a field for the quotient axiom, which says the representative function is a section of the canonical surjection. An instance of the quotient interface is called a quotient structure.

As we show in Figure 6.2, we build the interface following the same layout that the algebraic hierarchy has (see Chapter 2). Indeed we package the operators `pi` and `repr` and the axiom `reprK` in the *class* record, which the interface includes.

More precisely, the definition of the quotient interface is the following.

```
Structure quotType (T : Type) := QuotType {
  quot_sort :> Type;
  quot_class : quot_class_of quot_sort
}.

Record quot_class_of (T Q : Type) := QuotClass {
  repr : Q -> T;
  pi : T -> Q;
  reprK : forall x, pi (repr x) = x
}.
```

[1] The name `reprK` comes from a standard convention in the SSReflect library to use the suffix "K" for cancellation lemmas.

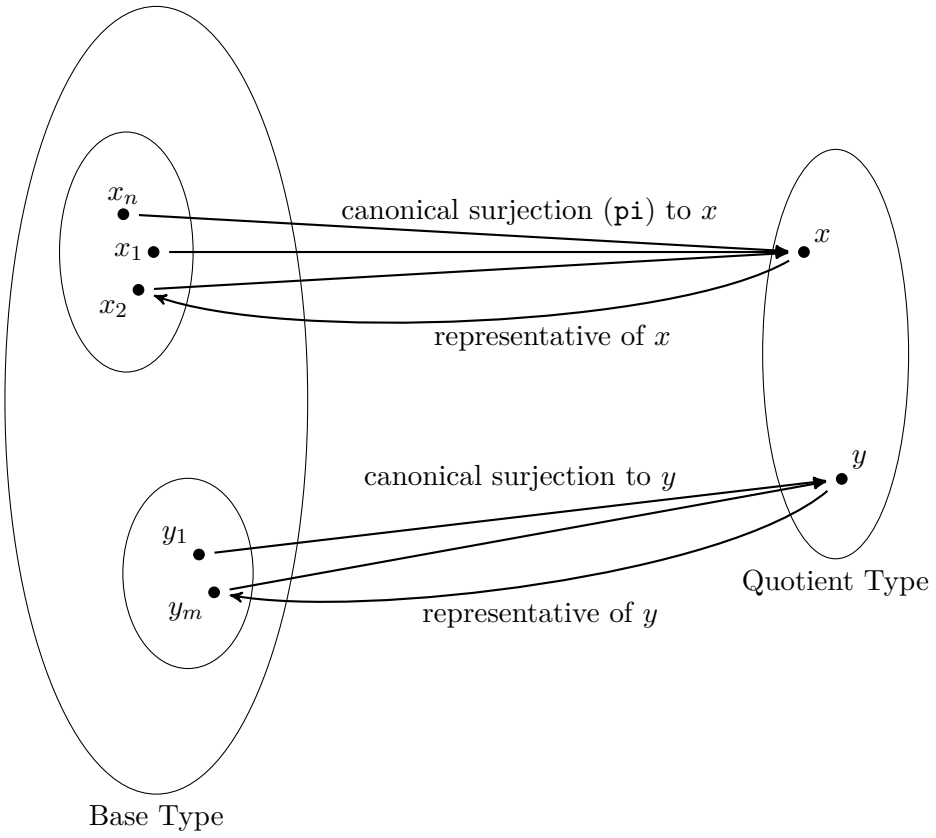


Figure 6.1: Quotients without equivalence relation

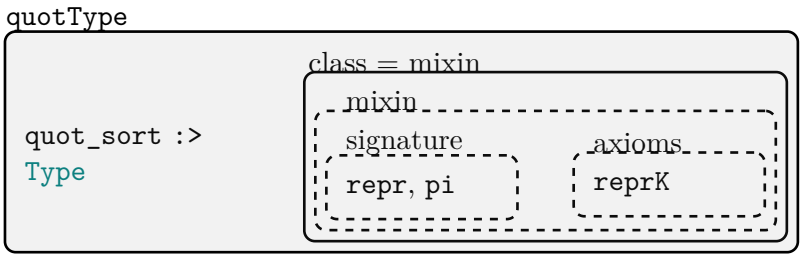


Figure 6.2: Quotient interface

Example

Let us define the datatype `int` of integers as the quotient of pairs of natural numbers by the diagonal. In other words, integers are the quotient of $\mathbb{N} \times \mathbb{N}$ by the equivalence relation

$$((n_1, n_2) \equiv (m_1, m_2)) \quad \hat{=} \quad (n_1 + m_2 = m_1 + n_2).$$

Now, we explicitly define the type of canonical represents: pairs of natural numbers such that one of them is zero. For example, the integer zero is represented by $(0, 0)$, one by $(1, 0)$ and minus one by $(0, 1)$.

```
Definition int_axiom (z : nat * nat) := (z.1 == 0) || (z.2 == 0).
Definition int := {z | int_axiom z}.
```

Example (continued)

We then define the particular instances `reprZ` of `repr` and `piZ` of `pi` and we show that `reprZ` is indeed a section of `piZ`:

```
Definition reprZ : int -> nat * nat := @projT1 _ _.

Lemma sub_int_axiom x y : int_axiom (x - y, y - x).

Definition piZ (x : nat * nat) : int :=
  @exist _ _ (x.1 - x.2, x.2 - x.1) (sub_int_axiom _ _).

Lemma reprZK x : piZ (reprZ x) = x.
```

Now, we pack together `reprZ`, `piZ` and `reprZK` into the quotient class, and the in the quotient structure `int_quotType`.

```
Definition int_quotClass := QuotClass reprZK.
Definition int_quotType := @QuotType int int_quotClass.
```

We created a data type `int` which is the candidate for being a quotient, and a structure `int_quotType` which packs `int` together with the evidence that it is a quotient.

Remark

Please note that this is not the way we formalized the type `int` of integers for the SSReflect library, as seen in Section 4.4.1.

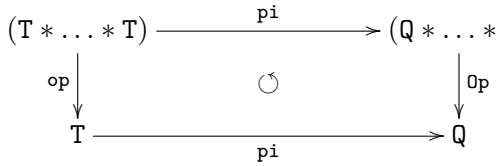
Recovering an equivalence and lifting properties

The existence of a quotient type `Q` with its quotient structure `qT` over the base type `T` induces a natural equivalence over `T`: two elements `(x y : T)` are equivalent if `(pi qT x = pi qT y)`. We mimic the notation of the SSReflect library for equivalence in modular arithmetic on natural numbers, which gives us the following notation for equivalence of `x` and `y` of type `T` modulo the quotient type `qT`:

```
x = y %[mod qT]
```

We say an operator on `T` (*i.e.* a function which takes its arguments in `T` and outputs an element of `T`) is compatible with quotienting if given two lists of arguments which are pairwise equivalent, the two outputs are also equivalent. In other words, an operator is compatible with quotienting if it is constant on each equivalence class, up to equivalence.

When an operator `op` on `T` is compatible with quotienting, it has a *lifting*, which means there exists an operator `Op` on the quotient type `Q` such that following diagram commutes:



The canonical surjection is a morphism for this operator.

For example, a binary operator `(Op : Q -> Q -> Q)` is a lifting for the binary operator `(op : T -> T -> T)` with regard to the quotient structure `qT` as soon as the canonical surjection `(pi qT : T -> Q)` is a morphism for this operator:

```
forall x y, pi qT (op x y) = Op (pi qT x) (pi qT y) :> Q
```

which can be re-expressed in a standardized form for morphisms in SSReflect:

```
{morph (pi qT) : x y / op x y >-> Op x y}
```

Example

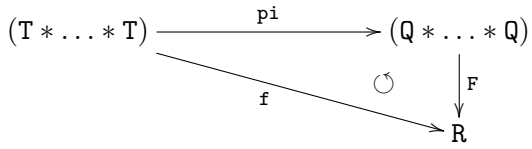
Let us define the add operation on `int` as the lifting of the point-wise addition on pairs of natural numbers.

```
Definition add x y := (x.1 + y.1, x.2 + y.2).
Definition addz X Y := \pi_int (add (repr X) (repr X)).
Lemma addz_compat : {morph \pi_int : x y / add x y >-> addz x y}.
```

We recall that the statement of `addz_compat` can be read as follows:

```
forall x y, \pi_int (add x y) = addz (\pi_int x) (\pi_int y)
```

Similarly, given an arbitrary type `R`, we say that a function with values in `R` is compatible with quotienting if it is constant on each equivalence class. When a function `f` with arguments in `T` and values in `R` is compatible with quotienting, it has a *lifting*, which means there exists an operator `F` with arguments in the quotient type `Q` and values in `R` such that the following diagram commutes:



The canonical surjection is a morphism for this function.

For example, a binary function `(F : Q -> Q -> R)` is a lifting of a binary function `(f : T -> T -> R)` if:

```
forall x y, (f x y) = F (pi qT x) (pi qT y) :> R
```

which can be re-expressed in a standardized form for morphisms in SSReflect:

```
{mono (pi qT) : x y / f x y >-> F x y}
```

Remark

It is interesting to remark for further generalization that the compatibility and lifting of operators is a particular case of the compatibility and lifting of functions, where we set `f` $=$ `pi` $\circ$ `op`.

Inference of quotient structures

Given a base type `T`, we say `Q` is a quotient type for `T` if there is a quotient structure `qT` which `Q` is the `quot_sort` field of, which means `(quot_sort qT)` evaluates to `Q`. In practice, given `x`, in `Q` we want to be able to write `(repr x)`, but such a statement would be ill-typed.

Technical remark

This impossibility comes from the fact `repr` has an implicit argument which must have type `quotType`. The expanded form for `(repr x)` is `(@repr` $?_{\texttt{quotType}}$ `x)`, where `x` must have type `(quot_sort` $?_{\texttt{quotType}}$`)`. But if `x` has type `Q`, the type inference algorithm encounters the unification problem

$$(\texttt{quot_sort}\ ?_{\texttt{quotType}}) \quad \equiv \quad \texttt{Q}$$

which it cannot solve without an hint, although we known the solution is `qT`.

However, it is possible to make the system type this statement anyway, by providing the information that `qT` is a canonical structure for `Q`.

Example (continued)

We make `int_quotType` the canonical quotient structure for the quotient type `int` for by using the following Coq vernacular command:

```
Canonical int_quotType.
```

Now, given `x` of type `int`, the system typechecks `(repr x)` as an element of `(nat * nat)`, as expected.

Since a quotient structure is canonically attached to every quotient type, we may also simplify the use of `pi`. Indeed, for now, `pi` has the following type.

```
forall (T : Type) (qT : quotType T), T -> quot_sort qT
```

Hence, `(@pi _ qT)` has type `T -> Q`, but it is not possible to use `(@pi _ Q)` to refer to this function. To circumvent this problem we provide a notation `\pi_Q` which gives exactly `(@pi _ qT)` where `qT` is the canonical quotient structure attached to `Q` if it exists (otherwise, the notation fails). This notation uses a standard trick from the SSReflect library to let the system infer `qT` automatically.

Technical remark

To achieve this, we define an alternative version for `pi` we call `pi_of`.

```
Definition pi_of T (qT : quotType T)
  (_ : phant (quot_sort qT)) : quot_sort qT := @pi T qT.
Notation "\pi_ Q" := (@pi_of _ _ (Phant Q)) : quotient_scope.
```

This alternative version of `pi` uses phantom types from SSReflect library, which is a simple inductive with one constructor, no argument and one type parameter.

```
Inductive phant (T : Type) := Phant.
```

The essential property of phantoms is that `(Phant Q)` has type `(phant Q)`. Hence, writing `(\pi_Q)` triggers the unification problem

$$\texttt{phant}\ (\texttt{quot_sort}\ ?_{\texttt{quotType}}) \quad \equiv \quad \texttt{phant Q}$$

on the type of third argument of `pi_of`. As a consequence, if `Q` has a canonical quotient structure `qT`, then `\pi_Q` resolves to `(@pi_of T qT (Phant Q))` which is convertible to `(@pi T qT)`.

Example

`\pi_int` has type `(nat * nat -> int)` and is in fact `(@pi int_quotType)`

We also adapted the notation for equivalence modulo quotient, so that we can provide `Q` instead of `qT`, as follows:

```
x = y %[mod Q]
```

This is a notation for `(\pi_Q x = \pi_Q y)`.

Exploiting compatibility with quotients

Technical remark

All this section is quite technical.

When an operation (or a function) is compatible with quotienting, we can forge the lifting by hand by composing the initial operation with `pi` and `repr`. In this case the canonical surjection is indeed a morphism for the operator.

Then we want to show elementary properties on the lifting, and those can often be derived from the properties of the initial operation. Thanks to the compatibility lemma, it is easy to go back and forth between the operation and its lifting by making `pi` and the operation commute.

Example

We show that zero is a neutral element on the right for addition:

```
Definition zeroz := \pi_int (0, 0).
Lemma add0z x : addz zero x = x.
```

Then, by using `reprK`, the statement of `add0z` is equivalent to:

```
addz (\pi_int (0, 0)) (\pi_int (repr x)) = (\pi_int (repr x))
```

which can be solved using `addz_compat`.

However, it becomes more complicated to control where rewriting must happen when faced with more complex expressions involving lifted operators.

Example

```
Lemma addzA x y z : addz x (addz y z) = addz (addz x y) z.
```

By using `reprK`, the left hand side is equivalent to:

```
addz (\pi_int (repr x)) (\pi_int (add (repr (\pi_int x)) (repr
     (\pi_int z))))
```

But now, the right hand side of `addz_compat` has an invisible instance.

In order to save the user from the need to use a chain of rewriting rules of the form `op_compat`, we introduce an automated mechanism to globally turn an expression on the quotient into an expression on the base type. For this, an operation `Op` (respectively a function `F`) which is a lifting has to be recognized automatically. We must register in some way that the value associated with it is `(\pi_Q (op x y))` (respectively `(f x y)`). For this purpose, we define a structure `pi_morph`:

```
Record pi_morph Q (u : Q) := PiMorph {pi_op : Q ; _ : u = pi_op}.
```

```
Lemma piE (Q : Type) (u : Q) (m : pi_morph u) : pi_op m = u.
Proof. by case: m. Qed.
```

The type parameter `u` of the structure should be the data to be inferred (*i.e.* `(\pi_Q (op x y))` for a binary operator) while the content of the field `pi_op` should be the information present in the goal (*i.e.* `(Op (\pi_Q x)(\pi_Q))` for a binary operator), such that rewriting using `piE` would cause the canonical surjections `\pi_Q` to go from the leafs to the root of the syntactic tree.

We declare an instance for `op`:

```
Canonical op_pi_morph (x y : T) (qT : quotType T)
 (X : pi_morph (\pi_qT x)) (Y : pi_morph (\pi_qT y)) :
 pi_morph (\pi_qT (op x y)) := @PiMorph (Op X Y) proof_it_works
```

Now, we must declare that any term of the form `(\pi_Q x)` has a trivial `pi_morph` structure (where both `u` and `pi_op` are `(\pi_Q x)`):

```
Canonical pi_morph_pi T (qT : quotType T) (x : T) :=
   @PiMorph _ (\pi_qT x) (\pi x) (erefl _).
```

Example

```
Lemma addz_pi (x y : T)
  (X : pi_morph (\pi_Q x)) (Y : pi_morph (\pi_Q y)) :
    \pi_int (add x y) = addz X Y.

Canonical addz_pi_morph (x y : T)
  (X : pi_morph (\pi_Q x)) (Y : pi_morph (\pi_Q y)) :
    pi_morph (\pi_Q (op x y)) := @PiMorph (Op X Y) (addz_pi X Y).
```

By declaring `addz_pi_morph` as canonical, we can now use `piE` to rewrite an expression of the form `(addz` $\tilde{x}$ $\tilde{y}$`)` into `(\pi_int (add x y))`, where $\tilde{x}$ and $\tilde{y}$ are arbitrarily complicated expression that can be canonically recognized as being equal to respectively `(\pi_int x)` and `(\pi_int y)`.

In the examples above, we had to define the quotient by hand. We could expect a generic construction of a quotient by a given equivalence relation. We now deal with this deficiency.

6.3 Quotient by an equivalence relation

Until now we have shown a quotient interface with no equivalence in its signature, and a notion of equivalence which is defined from the quotient. Now, we explain how to instantiate the quotient interface by quotienting a type by an equivalence relation. Given a type `T` and an equivalence relation `equiv`, we have to find a data type representing the quotient *i.e.* such that each element is an equivalence class.

A natural candidate to represent equivalence classes is the Σ-type of predicates that characterize a class. The elements of a given equivalence class are characterized by a predicate `P` that satisfies the following `is_class` property :

```
Definition is_class (P : T -> Prop) :=
  (forall x y, P x -> (P y <-> equiv x y)) /\ (exists x, P x).
```

Thus, we could define the quotient as follows.

```
Definition quotient := {P : T -> Prop | is_class P}.
```

However, because Coq equality is intentional, two predicates which are extensionally equal (*i.e.* equal on every input) may be different, and if that was not enough, the proof that a given predicate is a class is not unique either.



Remark

We can notice that equivalence classes are all an `(equiv x)` for some `x` (up to equivalence), indeed:

```
Lemma class_is_equiv P : is_class P <->
  exists x, (forall y, P y <-> equiv x y).
```

Things are different when the equivalence is decidable (with return type `bool`). In this case, there is only one proof of `(equiv x y)` for each `x` and `y`, because of Boolean proof irrelevance, as explained in Section 1.2. However, there are many possible `(equiv y)` for elements `y` such that `(equiv x y)`. This is not enough to make `quotient` a quotient type.

6.3.1 Quotient of a choice structure

Given a base type `T` equipped with a choice structure (in the sense of Chapter 2) and a decidable equivalence relation `(equiv : T -> T -> bool)`, it becomes possible to build a quotient type. The construction is slightly more complicated that the one above.

For each class we can choose an element `x` in a canonical fashion, using the following `canon` function:

```
Lemma equiv_exists (x : T) : exists y, (equiv x) y.
Proof. by exists x; apply: equiv_refl. Qed.

Definition canon (x : T) := xchoose (equiv_exists x).
```

We recall that `xchoose` takes a proof of existence of an element satisfying a predicate (here the predicate `(equiv x)`) and returns a witness which is unique, in the sense that two extensionally equal predicates lead to the same witness. This happens for example with the two predicates `(equiv x)` and `(equiv y)` when `x` and `y` are equivalent: the choice function will return the same element `z` which will be equivalent both to `x` and `y`. Such a canonical element is a unique representative for its class.

Hence, the type formed with canonical elements can represent the quotient.

```
Record equiv_quot := EquivQuot {
  erepr : T;
  erepr_canon : canon erepr == erepr
}.
```

The representative function is trivial as it is exactly the projection `erepr` on the first field of the Σ-type `equiv_quot`. However, a little more effort is needed to build the canonical surjection. Indeed we first need to prove that `canon` is idempotent.

```
Lemma canon_id (x : T) : canon (canon x) == canon x.
Definition epi (x : T) := @EquivQuot (canon x) (canon_id x).
```

Finally, we need to prove that the canonical surjection `epi` cancels the representative `erepr`:

```
Lemma ereprK (u : equiv_quot T) : epi (erepr u) = u.
```

The proof of `ereprK` relies on the proof irrelevance of Boolean predicates.

Proof. Two elements of `equiv_quot` are equal if and only if their first projection `erepr` are equal, because the second field `erepr_canon` of `equiv_quot` is a Boolean equality, and has only one proof. Thanks to this `(epi (erepr u))` and `u` are equal if and only if `(erepr (epi (erepr u))` is equal to `(erepr u)`. But by definition of `epi`, `(erepr (epi (erepr u))` is equal to `(canon (erepr u))`, and thanks to the property `(erepr_canon u)`, we get that `(canon (erepr u))` is equal to `(erepr u)`, which concludes the proof. □

We then package everything into a quotient structure:

```
Definition equiv_quotClass := QuotClass ereprK
Canonical equiv_quotType := @QuotType equiv_quot equiv_quotClass.
```

We declare this structure as canonical, so that any quotient by an equivalence relation can be recognized as a canonical construction of quotient type.

However, we omitted to mention that the proof of `canon_id` and hence the code of `epi` requires a proof that `equiv` is indeed an equivalence relation. In order to avoid adding unbundled side conditions ensuring `equiv` is an equivalence relation, we define an interface for equivalence relations which coerces to binary relations:

```
Structure equiv_rel := EquivRelPack {
  equiv_fun :> rel T;
          _ : reflexive equiv
          _ : symmetric equiv
          _ : transitive equiv
}.
```

The whole development about `equiv_quot` takes `(equiv : equiv_rel T)` as a parameter.



Technical remark

Given an equivalence relation `equiv` on a choice type `T`, we made a notation `{eq_quot equiv}` to create a quotient by inferring both the equivalence structure of `equiv` and the choice structure of `T` and applying the latter construction.

We recall that we defined the equivalence induced by the quotient by saying `x` and `y` are equivalent if `(\pi_Q x = \pi_Q y)`, where `Q` is the quotient type. We refine the former notation for equivalence modulo `Q` to specialize it to quotients by equivalence, as follows.

```
x = y %[mod_eq equiv]
```

In the present situation, it seems natural that this induced equivalence coincides with the equivalence by which we quotiented.

```
Lemma eqmodP x y : reflect (x = y %[mod_eq equiv]) (equiv x y).
```

Example

Let us redefine once again `int` as the quotient of $\mathbb{N} \times \mathbb{N}$ by the equivalence relation $((n_1, n_2) \equiv (m_1, m_2))$ defined by $(n_1 + m_2 = m_1 + n_2)$.
In this second version, we directly perform the quotient by the relation, so we first define the equivalence relation.

```
Definition equivnn (x y : nat * nat) := x.1 + y.2 == y.1 + x.2.

Lemma equivnn_refl : reflexive equivnn.
Lemma equivnn_sym : symmetric equivnn.
Lemma equivnn_trans : transitive equivnn.

Definition equivnn_rel :=
  EquivRel equivnn_refl equivnn_sym equivnn_trans.
```

Then `int` is just the quotient by this equivalence relation.

```
Definition int := equiv_quot equivnn_rel.
```

This type can be equipped with a quotient structure by repackaging the quotient class of `equiv_quotType equivnn_rel` together with `int`.

6.3.2 Quotient of type with an explicit encoding to a choice structure

We will need in Section 7 to quotient, by a decidable equivalence, a type which is not a choice type.

We say a type `T` with a equivalence relation `equivT` is explicitly encodable to a type `C` if there exists two functions `(T2C : T -> C)` and `(C2T : C -> T)` such that the following *coding property* holds:

```
forall (x : T), equivT (C2T (T2C x)) x.
```

Remark

Here `T` can be seen as a setoid, and the coding property can be interpreted as: `C2T` cancels `T2C` in the setoid `T`.

The function `(T2C : T -> C)` is called the *encoding* function because it codes an element of `T` into `C`. Conversely, the function `(C2T : C -> T)` is called the *decoding* function. The coding property expresses that encoded elements can be decoded properly.

There is no notion of equivalence on the coding type `C` yet, but we can provide one using the equivalence induced by `T`. Thus, we define `equivC` by composing `equivT` with `C2T`.

```
Definition equivC x y := eT (C2T x) (C2T y).
```

When `equivT` is Boolean, `equivC` is. Since `C` is a choice type and `equivC` is a decidable equivalence on this choice type, we can reproduce the exact same construction of quotient as in Section 6.3.1, so that we get `ereprC : equiv_quot -> C`, `epiC: C -> equiv_quot` and a proof of cancellation `ereprCK`. Now we can compose these operators with `T2C` and `C2T`.

```
Definition ereprT (x : equiv_quotient) : T := C2T (ereprC x).
Definition epiT (x : T) : equiv_quotient := epiC (T2C x).
```

And we can prove the cancellation lemma `ereprTK` using `ereprCK` and the coding property.

```
Lemma ereprTK (x : equiv_quotient) : epiT (ereprT x) = x.
```

Finally, we have everything we need to create a quotient type, like in Section 6.3.1.

6.4 Related work on quotient types

Given a base type `(T : Type)` and an equivalence `(equiv : T -> T -> Prop)`, the Coq interface below is due to Laurent Chicli, Loïc Pottier and Carlos Simpson [20], following studies from Martin Hofmann [55]. It sums up the desired properties of a the quotient type: its existence, a surjection from the base type `T` to it, and a way to lift to the quotient functions that are compatible with the equivalence `equiv`.

```
Record type_quotient (T : Type) (equiv : T -> T -> Prop)
  (Hequiv : equivalence equiv) := {
  quo :> Type;
  class :> T -> quo;

  quo_comp : forall (x y : T), equiv x y -> class x = class y;
  quo_comp_rev : forall (x y : T), class x = class y -> equiv x y;
  quo_lift : forall (R : Type) (f : T -> R),
     compatible equiv f -> quo -> R;
  quo_lift_prop :
     forall (R : Type) (f : T -> R) (Hf : compatible equiv f),
     forall (x : T), (quot_lift Hf \o class) x = f x;
  quo_surj : forall (c : quo), exists x : T, c = class x
}.
```

where `\o` is the infix notation for functional composition and where `equivalence` and `compatible` are predicate meaning respectively that a relation is an equivalence (reflexive, symmetric and transitive) and that a function is constant on each equivalence class. We believe[2] they are defined as below:

```
Definition equivalence (T : Type) (equiv : T -> T -> Prop) :=
  reflexive equiv /\ symmetric equiv /\ transitive equiv.
Definition compatible (T R : Type) (equiv : T -> T -> Prop)
   (f : T -> R) := forall x y : T, equiv x y -> f x = f y.
```

Once this `type_quotient` defined, they [20] add the existence of the quotient as an axiom.

```
Axiom quotient : forall (T : Type) (equiv : T -> T -> Prop)
   (p:equivalence R), (type_quotient p).
```

Although this axiom is not provable in the type theory of Coq, its consistency in the Calculus of Construction has been proved in [55]. The construction of this interface

[2] No definitions for `equivalence` or `compatible` are explicitly given in [20].

was made in order to study the type theory of Coq augmented with quotienting. This is not our objective at all. First, we want to keep the theory of Coq without modification, so quotient types do not exist in general. Second, we create an interface to provide practical tools to handle quotients types that do exist.

The reader may notice that here `quo` plays the role of our `quot_sort` and `class` the role of `pi` of our interface. The combination of `repr` and `reprK` is a skolemized version of `quo_surj`.



Technical remark

This is not exactly the case, because `quot_surj` is a `Prop` existential, which unlike existentials in `Type` cannot be extracted to a function `repr` which has the property `reprK`. This was already observed [20] in the study of the consistency of Coq with variants of `type_quotient`.

However, the parameters about `equiv` and properties about the lifting of morphism disappear completely in our interface, because they can all be encoded as explained in Section 6.2.

Example

For example, `quo_lift` can be encoded like this:

```
Definition new_quo_lift (T R : Type) (qT : quotType T)
  (f : T -> R) (x : Q) := f (repr x)
```

Note that the precondition `(compatible equiv f)` was not needed to define the lifting `new_quo_lift`. Only the property `quo_lift_prop` still needs the precondition.

Our approach can also be compared to Normalized Types [31]. The function `pi` can be seen as a user defined normalization function inside Coq.

7 Construction of the real closure as a type

While in Chapter 5.2 we constructed the algebraic Cauchy reals, we now focus on the construction of a datatype for algebraic numbers, where each element represents a distinct algebraic number.

In this chapter we show that the algebraic Cauchy reals satisfy the requirements for building a quotient type as described in Section 6.3.2. For that purpose we explain in Section 7.1.1 the decoding and the encoding in Section 7.1.2. Finally, in Section 7.2 we form the quotient type `(alg F)` of algebraic numbers over `F`, and we show that the operator `(alg : archiFieldType -> Type)`, which canonically outputs an `archiFieldType` is involutive (in the sense that applying twice gives a field which is isomorphic to the single application) and is hence a closure operator. From this last fact we derive the real closed field axiom for `(alg F)`, making this field a real closed field.

7.1 Algebraic numbers have an explicit encoding to a choice type

In order to get the type of real algebraic numbers, we should quotient the type of algebraic Cauchy reals by the setoid equality. In order to apply the construction we detail in Section 6.3.2, we explicit an encoding from algebraic Cauchy reals to a choice type, which we name `algdom` and call real algebraic domain. The type `algdom` not only serves as an encoding of `algcreal` in order to forge the quotient, but also as finite descriptions for real algebraics.

```
Record algdom := AlgRealDom {
  annul_algdom : {poly F};
  center_alg : F;
  radius_alg : F;
  _ : monic annul_algdom;
  _ : annul_algdom.[center_alg - radius_alg]
      * annul_algdom.[center_alg + radius_alg] <= 0
}.
```

An element `(AlgRealDom P c r monic_P chg_sign_P)` of `algdom` represents one of the roots of the polynomial `P` in the interval `[c - r, c + r]`, with a proof `monic_P` that `P` is monic and a proof `chg_sign_P` that `P` changes sign on the interval. Note that the unicity of the root this representation designates is not guaranteed by the axioms of this record. However, the decoding procedure given in Section 7.1.1 is a (deterministic) function from `algdom` to algebraic Cauchy reals. Only one root can be selected by this procedure, and this is how the ambiguity vanishes. Naturally, changing the polynomial or the interval of an element from the algebraic domain may change the root it designates, but it does not matter for our purpose.

This datatype is only using elements of F – the list of coefficients of the polynomial, together with the center and the radius of the interval – and two proofs. Thanks to the proof irrelevance of Boolean propositions (seen in Section 1.2), we can build a bijection between `algdom` and the type of sequences of elements of F. Hence, `algdom` inherits the `choiceType` structure of `(seq F)`. We also notice that `algdom` is countable as soon as F is. This fact was not obvious for the setoid of algebraic Cauchy reals. The quotient type will also inherit the `choiceType` structure and will be countable if F is.

This representation gives a finite description of algebraic numbers. For example, we could represent $\sqrt{2}$ as the pair $(X^2 - 2, [0, 2])$, and more generally $\sqrt{x}$ as the pair $(X^2 - x, [0, x])$.

We now show that `algdom` is an explicit encoding of algebraic Cauchy reals. In order to do so, we build an encoding and a decoding function as shown in Figure 7.1.

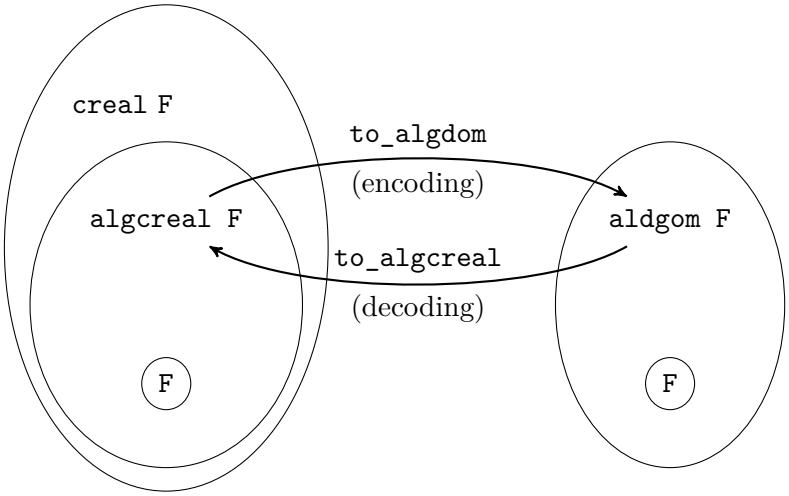


Figure 7.1: Encoding algebraic numbers as a choice structure

7.1.1 Decoding to algebraic Cauchy reals

We build the decoding function.

```
Definition to_algcreal : algdom -> algcreal.
```

An element from the real algebraic domain is given by a monic polynomial P, a center c and a radius r such that $P(c-r)P(c+r) \leq 0$. The root we wish to select is in the interval $I = [c-r, c+r]$.

We decode an element from the real algebraic domain into an algebraic Cauchy real by dichotomy. We form the Cauchy sequence $x = (x_n)_n$, such that all the x_n are in the interval I and such that $P(x) \equiv 0$.

We proceed by induction on n to define the sequence x. It should satisfy the following invariant, which expresses that P must change sign on the interval of radius $2^{-n}r$ and centered in x_n:

$$H_n = P(x_n - 2^{-n}r)P(x_n + 2^{-n}r) \leq 0$$

The base case is trivial, it suffices to choose x_0 to be c. In the induction step, we pick either $x_n - 2^{-(n+1)}r$ or $x_n + 2^{-(n+1)}r$ to satisfy the invariant H_{n+1}.

7.1.2 Encoding of algebraic Cauchy reals

Now, we construct the encoding function, which is more difficult.

```
Definition to_algdom : algcreal -> algdom
```

It must satisfy the coding property as mentioned in Section 6.3.2:

```
Lemma to_algdomK x : to_algcreal (to_algdom x) == x.
```

Given an algebraic Cauchy real (x, P), we try to find a rational interval containing only one root, in order to be sure that the decoding procedure returns exactly the element equivalent to x. Moreover, the polynomial has to change sign between the bounds of the interval we are looking for. In order to ensure all that, we look for an interval on which P is monotone.

We reason by induction on the degree of P. The base case is trivial as P cannot be zero because it is monic. We start the inductive step by a discussion on the coprimality of P and its derivative P'.

Either P and its derivative P' are coprime, so there exist U and V such that $UP + VP' = 1$. Since $P(x)$ converges to 0, by taking i big enough, we get $P'(x_i) \geq \frac{1}{2\lceil V(x) \rceil}$. By taking a small enough interval $[a, b]$ containing x_n, we get that P is monotone on $[a, b]$ (thanks to the B_2 bound of Section 5.1.3).

Without loss of generality, we can suppose that P is increasing. Then we get $P(a) \leq P(x_i) \leq P(b)$ for all $i \geq n$. But $P(x_i)$ converges to 0, so $P(a) \leq 0 \leq P(b)$. Hence, the interval $[a, b]$ satisfies the requirements.

Or P and P' are not coprime, and since P cannot divide P' (because of their degrees), using Lemma 5.4, we can find a monic polynomial R which annihilates x but which degree is smaller than the one of P. We fall back to the study of (x, R), where the degree of R is strictly smaller that the one of R. The induction hypothesis applies and gives an interval which satisfies the requirements.

7.2 A quotient type for algebraic numbers

7.2.1 Construction of the quotient type

Let `F` be a discrete Archimedean field. Because `algcreal` is encodable to a choice structure, and because `eq_algcreal` is a decidable equivalence relation (see Section 5.2.1), we are exactly in the context of Section 6.3.2. Hence we can forge the

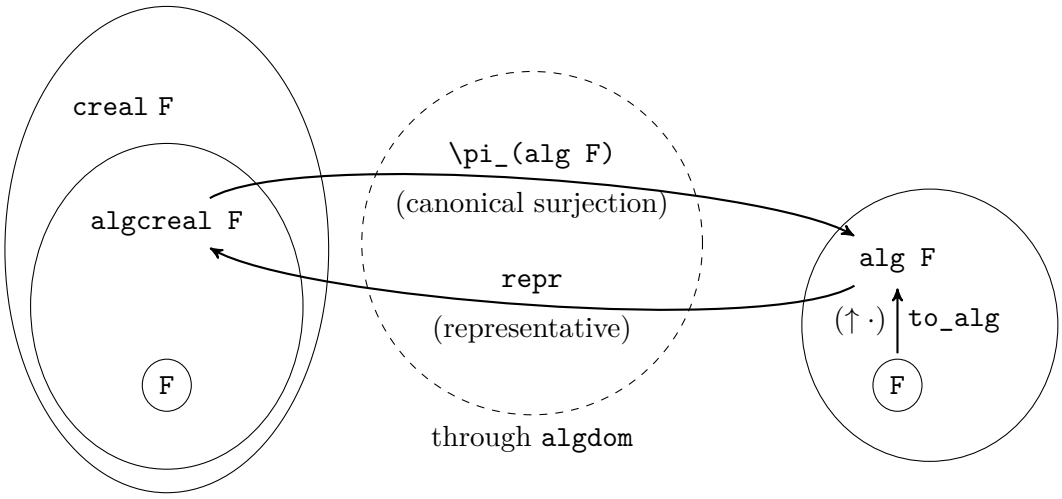


Figure 7.2: The construction of real algebraic numbers

type `alg` of real algebraic numbers by quotienting algebraic Cauchy reals by its setoid equality.

```
Definition alg := {quot_eq eq_algcreal}.
```

This type `alg` is the type of real algebraics over the base field `F`. When there is an ambiguity on the base field, we write `(alg F)`. We use the mathematical notation Alg_F to denote the type of real algebraics over F.

Remark

If `F` is countable, then `algdom` is also countable. Because we implemented quotienting using subtyping, `alg` is a subtype of `algdom` and is hence countable when `F` is.

We prove that arithmetic operations and the order relation are compatible with the quotient. This is a direct consequence of the morphism property of operations with regard to setoid equality, which we dealt with in Section 5.2.2. Thanks to this, we have liftings to algebraics for every arithmetic operation, and for the order relation. Moreover we can show that Alg_F has an Archimedean field structure. This means Alg can be seen as an operator from Archimedean fields to Archimedean fields.

We also build a function `(to_alg: F -> alg F)` which embeds any element c of F into Alg_F, by mapping c to the canonical surjection of the element $(c, (X - c))$ of `algcreal`. We then prove it is a field morphism and that this morphism is also compatible with comparison. Hence, it is a numeric field morphism. The mathematical notation for this function is $\uparrow$, because it goes up in Figure 7.2. This morphism can also be applied to a polynomial (by applying it to each coefficient).

We remark that by construction of `algdom`, the following property holds:

Lemma 7.1 (Weak intermediate value theorem for Alg_F)**.** *Given a polynomial P in $F[X]$ and two points $a < b$ in F such that $P(a) \leq 0 \leq P(b)$, there exists c in Alg_F such that c is in $[\uparrow a, \uparrow b]$ and $(\uparrow P)(c)$ is zero.*

7.2.2 Real algebraic numbers form a real closed field

We already know that Alg_F is an Archimedean field. The difficulty in proving Alg_F is a real closed field is to prove the intermediate value theorem for polynomials in $\mathrm{Alg}_F[X]$. Let us recall the statement of this property.

Definition: Intermediate value property for polynomials

Let R be a real field, we say the intermediate value property holds for polynomials in $R[X]$ if given a polynomial P in $R[X]$ and two points $a < b$ in R such that $P(a) \leq 0 \leq P(b)$, there exists c in R such that c is in $[a, b]$ and $P(c)$ is zero.

Now, let P be a polynomial in $\mathrm{Alg}_F[X]$ and a and b two elements of Alg_F such that $a < b$ and $P(a) \leq 0 \leq P(b)$. Let us show that there exist an real algebraic number c in Alg_F such that $c \in [a, b]$ and $P(c) = 0$.

Iteration of Alg.

Because Alg can be seen as a function from Archimedean fields to Archimedean fields, we can consider $\mathrm{Alg}_{\mathrm{Alg}_F}$. The weak intermediate value theorem for $\mathrm{Alg}_{\mathrm{Alg}_F}$ applied to P, a and b gives the existence of γ in $\mathrm{Alg}_{\mathrm{Alg}_F}$ such that γ is in $[\uparrow a, \uparrow b]$ and $(\uparrow P)(\gamma)$ is zero. It suffices we find c in Alg_F such that $\uparrow c = \gamma$ to conclude. Indeed, such a c would satisfy $\uparrow c \in [\uparrow a, \uparrow b]$ and $(\uparrow P)(\uparrow c) = 0$, and because $(\uparrow \cdot)$ is a numeric field morphism, we would get $c \in [a, b]$ and $P(c) = 0$.

For this, it suffices we find function $\downarrow$: $\mathrm{Alg}_{\mathrm{Alg}_F} \to \mathrm{Alg}_F$ (see Figure), such that

$$\forall \xi \in \mathrm{Alg}_{\mathrm{Alg}_F}, \quad \uparrow (\downarrow \xi) = \xi,$$

because then, we would simply have to pose $c = \downarrow \gamma$. In Coq, we call this function `from_alg`. The existence of such a function means that Alg is in fact a closure operator.

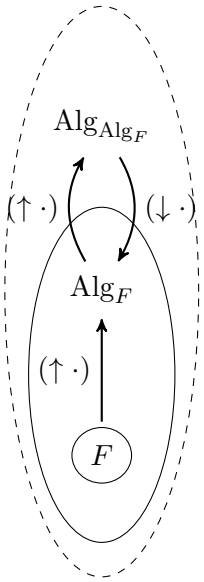


Figure 7.3: Alg is the real algebraic closure

Let ξ be in $\mathrm{Alg}_{\mathrm{Alg}_F}$, and let us build $(\downarrow \xi)$. By transforming ξ in an algebraic Cauchy real we get a Cauchy sequence x in $\mathrm{Alg}_F^{\mathbb{N}}$, and a polynomial $P \in \mathrm{Alg}_F[X]$, such that

$$\xi = \pi_{\mathrm{Alg}_{\mathrm{Alg}_F}}(x, P).$$

For all n in $\mathbb{N}$, each element x_n is an algebraic of Alg_F and there exists a Cauchy sequence $(a_{n,k})_{k\in\mathbb{N}}$ in $F^{\mathbb{N}}$ and a polynomial A_n in $F[X]$ such that

$$x_n = \pi_{\mathrm{Alg}_F}(a_n, A_n).$$

In fact we can even chose x such that $|x_{n+1} - x_n| < 2^{-(n+1)}$ in Alg_F. Now, the sequence $y = (a_{n,n})_n$ is a Cauchy sequence such that $\uparrow y = \xi$. Hence, the Cauchy real underlying $(\downarrow \xi)$ is y. It suffices to find a polynomial that cancels y to conclude.

Polynomial annihilating the algebraic Cauchy real y.

We must find a polynomial $R \in F[X]$ which annihilates y. The coefficients p_i of P form a finite set of values in the field extension Alg_F of F, so we can apply the primitive element theorem to find an element $\beta \in \mathrm{Alg}_F$, whose annihilating polynomial is Q of degree $q+1$ such that for all i, p_i is in the simple extension $F[\beta]$. We can then re-factorize P as

$$P = \sum_{l=0}^{q} \beta^l P_l.$$

Let us consider the resultant

$$R(Y) = \mathrm{Res}_X\left(\sum_{l=0}^{q} X^l P_l(Y), Q(X)\right).$$

We now show that it is nonzero and that it annihilates x.

R is nonzero. Let us suppose R is zero and find a contradiction. The property of Bézout gives $U, V \in F[X]$ such that U is nonzero, $\deg_X(U) < \deg(Q)$ and

$$U(X,Y)\sum_{l=0}^{q} X^l P_l(Y) = V(X,Y)Q(X).$$

Then by embedding in Alg_F and evaluating at $X = \beta$ we get $U(\beta, Y)P(Y) = 0$. But $P \neq 0$, thus $U(\beta, Y) = 0$. Then by taking the Y-leading coefficient $u(X)$ of $U(X,Y)$ we get

$$u(\beta) = 0 \quad \text{and} \quad u \in F[X] \quad \text{and} \quad u \neq 0 \quad \text{and} \quad \deg(u) < \deg(Q).$$

This gives a polynomial u annihilating β of degree smaller than the one of Q, and we can proceed by induction on the degree of Q.

R annihilates y. We have

$$R(a_{n,n}) = U(\beta_m, a_{n,n})\left(\sum_{l=0}^{q} \beta_m^l P_l(a_{n,n})\right) + V(\beta_m, a_{n,n})Q(\beta_m).$$

and we notice that the right hand side converges to zero when m and n grow.

8 Discussion on the algebraic closure

The final step in the construction of algebraic numbers is the construction of the algebraically closed field of complex algebraic numbers. We chose to construct the algebraic closure of an arbitrary closed field R, which we define simply as $C = R[i] = R[X]/(X^2+1)$ and which can be represented by R^2 with the appropriate operations. The main difficulty is to prove that the C we obtain is indeed algebraically closed (*i.e.* every polynomial of $C[X]$ has a root in C), which is a generalization of the well-known fundamental theorem of algebra (FTA, also known as d'Alembert-Gauss theorem). In Section 8.1 we study the alternative definitions of real closed fields, which include a proof of the fundamental theorem of algebra. We present a snapshot of the work that had to be done prior to a Coq formalization, although it is not the proof that has been retained for the formalization. In Section 8.2 we describe the alternative path we took to formalize the fundamental theorem of algebra in Coq and we also mention other alternatives and other paths we could have taken, using other parts of our work.

8.1 Equivalent definitions for real closed fields

Let R be a real field, according to Theorem 2.11 from [5], the following assertions are equivalent.

(a) (1) Any polynomial of $R[X]$ of odd degree has a root in R
    (2) For all $x \geq 0$, there exists some y such that $y^2 = x$

(b) The field $R[i]$ is algebraically closed (where i is a root of $X^2 + 1$)

(c) the intermediate value theorem holds for polynomials in $R[X]$

(d) R has no non-trivial real algebraic extension, that is there is no real field R_1 that is algebraic over R and different from R.

Figure 8.1 represents the implications proved in [5]. Note that not all the proofs given in this book are constructive. Let us first discuss what we need in this equivalence, then we will provide a constructive proof for what we need.

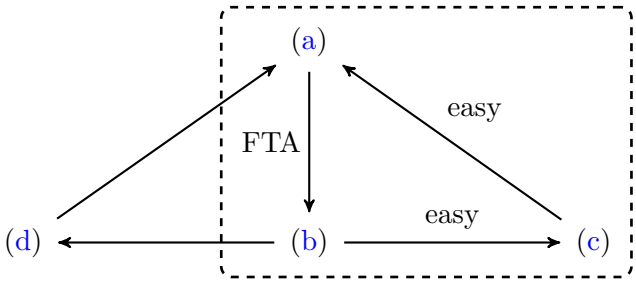


Figure 8.1: Equivalent definitions for real closed fields

The statement (a) is the original definition of real closed fields by Tarski [83], while (c) was introduced as an alternative definition in [5] because the proofs of quantifier elimination they use strongly rely on the intermediate value property for polynomials. Because our notion of real field is different from the one in [5] (see Section 4.1.2 for the comparison and the reason for this choice), we will not discuss the implications involving (d) which are independent from the (a) – (b) – (c) loop anyway.

The definition we took for real closed fields back in Section 4.1 was definition (c), because it was the only choice that did not involve proving the fundamental theorem of algebra for proving quantifier elimination and for constructing real algebraic numbers. Indeed, while constructing real algebraic numbers we could have provided a proof of either (a) or (c) without using the fundamental theorem of algebra. And quantifier elimination could have relied either on (c) or on (b) without using the fundamental theorem of algebra.

The construction of algebraic numbers requires (b), hence we need to prove (c) $\Rightarrow$ (b), which can be obtained by composing (c) $\Rightarrow$ (a) with (a) $\Rightarrow$ (b). The implications (b) $\Rightarrow$ (c) and (c) $\Rightarrow$ (a) are easy and constructive. However, the implication (a) $\Rightarrow$ (b) as shown in [5] uses the existence of the algebraic closure of R. In fact, as far as we know, it only requires a splitting field for a polynomial in $R[X]$, but the existence of a splitting field is not valid constructively in general. Actually, (a) $\Rightarrow$ (b) is showing that C is a splitting field for any polynomial in $R[X]$, but we cannot use it in its own proof. With no algebraic closure, another way to construct a splitting field extension for a polynomial p is by forging a sequence $K_1, \ldots, K_n$ of extensions of K such that $K_1 = R$ and K_{j+1} is obtained by quotienting $K_j[X]$ by a non trivial irreducible factor of p seen as a polynomial of $K_j[X]$ and repeating this operation until p splits in K_n. Although there are algorithms to do this when R is the field of real algebraic numbers (see Section 8.2), there is in general no algorithm that factors a polynomial or tells whether it is irreducible.

There are not many constructive algebraic proofs of the fundamental theorem of algebra in the literature, even in constructive algebra books (such as [62]). Surprisingly, one of the few we could find before we wrote ours [24] (with Thierry Coquand) was the second proof by Gauss [38], who was actually criticizing Laplace [61] about his proof relying on the existence of a splitting field, which holds classically but the proof of which was only made possible a few years later, after Abel and Galois explicitly introduced the notion of irreducible polynomial.

We now present a classical proof and two constructive modifications of this proof in prevision of the Coq formalization. The first constructive proof is elementary

and relies only on symmetric polynomials. The second one uses the universal decomposition algebra. These proofs detail in a "Coq ready style" what can be found in [24].

Classical proof

We want to show (a) ⇒ (b). Let R be a real closed field and $C = R[i]$ where $i^2+1 = 0$. If p is in $C[X]$, then $p\bar{p}$ is in $R[X]$, and if $p\bar{p}$ has a root $x \in C$, then either $p(x)$ or $\bar{p}(x)$ is zero, *i.e.* x or $\bar{x}$ is a root of p. So it suffices to show that any polynomial p of $R[X]$ has a root in C. It is also easy to show that without loss of generality we can suppose that p is monic.

Let $p = X^m + a_{m-1}X^{m-1} + \ldots + a_1X + a_0$ be a monic polynomial of degree m in $R[X]$. We show by induction on $v_2(m)$ (the multiplicity of 2 as a factor of m) that p has a root in C.

The base case: $v_2(m) = 0$ means that m is odd, so using (a.1), the polynomial p has a root in R, and hence in C.

The induction step: Let L be a splitting field of p (this is the non constructive argument). The polynomial p factors over L and let us call $x_1, \ldots, x_n$ its roots in L, so that $p = \prod_{i\in m}(X - x_i)$. The key ingredient, introduced by Laplace [61] is to define, for u in R,

$$q_u = \prod_{i_1<i_2} (X - x_{i_1} - x_{i_2} - ux_{i_1}x_{i_2}) \in L[X].$$

Its degree is $m' = \frac{m(m-1)}{2}$, so that $v_2(m') = v_2(m) - 1$. Moreover, it is symmetric in the roots $x_1, \ldots, x_m$, so that in fact q_u is in $R[X]$. Now, by induction hypothesis we get a root $y_u \in C$ of q_u for each $u \in R$, thus there exists a pair $i_1 < i_2$ such that $y_u = x_{i_1} + x_{i_2} + ux_{i_1}x_{i_2}$.

If we consider $m' + 1$ distinct values of u, by the pigeonhole principle we know there exists a pair $u \neq v$ corresponding to the same pair $i_1 < i_2$. In other words, we have both:

$$\begin{cases} y_u = x_{i_1} + x_{i_2} + ux_{i_1}x_{i_2} \\ y_v = x_{i_1} + x_{i_2} + vx_{i_1}x_{i_2}. \end{cases}$$

By inverting the system, we get both $s = x_{i_1} + x_{i_2}$ and $p = x_{i_1}x_{i_2}$. Since (a.2) gives us square roots of positives in R, it also gives us square roots of elements of C[1] Thanks to this, we can put the polynomial $X^2 - sX + p$ in canonical form and get

$$\{x_{i_1}, x_{i_2}\} = \left\{ \frac{-s \pm \sqrt{\Delta}}{2} \right\} \quad \text{with } \Delta = s^2 - 4p \in C$$

Hence x_{i_1} and x_{i_2} are in C and are roots of p, which concludes the induction step. □

[1] Indeed, let a and b be in R, let ρ be $\sqrt{a^2 + b^2}$ and ε be the sign of b, then

$$a + ib = \left(\sqrt{\frac{\rho + a}{2}} + i\varepsilon\sqrt{\frac{\rho - a}{2}} \right)^2$$

Constructive proof, using symmetric polynomials

Since we cannot invoke the existence of a splitting field for p, we replace this argument by the study of a formal multivariate polynomial. Let us take over the proof directly in the inductive case.

We recall that $p = X^m + a_{m-1}X^{m-1} + \ldots + a_1X + a_0$ is a monic polynomial of degree m in $R[X]$. Let $M_R = R[X_1, \ldots, X_m]$ (resp. $M_C = C[X_1, \ldots, X_n]$) be the ring of multivariate polynomials with m indeterminates and coefficients in R (resp. in C). For all u in R, we define

$$Q_u = \prod_{i \in I}(X - X_{i_1} - X_{i_2} - uX_{i_1}X_{i_2}) \in M_R[X].$$

where I is the set of pairs $i = (i_1, i_2)$ such that $0 \leq i_1 < i_2 \leq m$.

Let $\Sigma_1, \ldots, \Sigma_m$ be a new set of indeterminates. Since Q_u is symmetric in the indeterminates, we know by the fundamental theorem of symmetric polynomials (see [63] for example), that there exists a polynomial $\tilde{Q}_u$ in $R[\Sigma_1, \ldots, \Sigma_m]$ we call the *lifting* of Q_u, *i.e.*

$$\tilde{Q}_u(\sigma_1, \ldots, \sigma_m) = Q_u(X_1, \ldots, X_n),$$

where $\sigma_i \in M_R$ is the i^{th} elementary symmetric function of the $X_1, \ldots, X_m$.

Now, for u in R, we define q_u by

$$q_u = \tilde{Q}_u\left(-a_{m-1}, a_{m-2}, \ldots, (-1)^{m-1}a_1, (-1)^m a_0\right) \in R[X].$$

It is a univariate polynomial in X with coefficients in R, and its degree is $m' = \frac{m(m-1)}{2}$ so by induction hypothesis it has a root y_u in C. Unlike in the classical proof, y_u (which is in R) cannot be identified to some $X_i + X_j + uX_iX_j$.

However, for u and v in R, the system

$$\begin{cases} \alpha + u\beta = y_u \\ \alpha + v\beta = y_v \end{cases}$$

can be inverted to get a solution $(\alpha_{u,v}, \beta_{u,v})$ in C^2 expressed in function of u, v, y_u and y_v. Then, following the same method as in the classical proof (which used a.2), we can find two elements $r_{u,v}$ and $s_{u,v}$ in C such that $r_{u,v} + s_{u,v} = \alpha_{u,v}$ and $r_{u,v}s_{u,v} = \beta_{u,v}$. Hence,

$$r_{u,v} + s_{u,v} + ur_{u,v}s_{u,v} = y_u \quad \text{and } r_{u,v} + s_{u,v} + vr_{u,v}s_{u,v} = y_v.$$

Remark

In the classical setting, the latter q_u (the one in this proof) is equal to our former q_u (from the classical proof) and the latter y_u is equal to our former $x_{i_1} + x_{i_2} + ux_{i_1}x_{i_2}$ for some $i_1 < i_2$. So if we take u and v in a set of size at least $m'+1$, there should exist u and v such that $r_{u,v}$ is a root of p in C.

We now pick a set U of $m' + 1$ distinct elements of R and we define z as follows.

$$z = \prod_{u,v \in U/\ u<v} p(r_{u,v}) \in C$$

We claim that $z = 0$, which would mean that there exists $u < v$ such that $r_{u,v}$ is a root of p. We show that z is zero by first defining

$$Z = \prod_{u,v \in U/\, u<v} P(r_{u,v}) \in M_C \quad \text{where } P = \prod_{l \in \{1,\dots,m\}} (X - X_l) \in M_R[X].$$

We remark that both P and Z are symmetric, and the lifting $\tilde{Z}$ of Z evaluates to z when instantiated in $(-a_{m-1}, a_{m-2}, \dots, (-1)^{m-1}a_1, (-1)^m a_0)$.

Now the **key fact** to prove is that a power of Z is in the ideal generated by the multivariate symmetric polynomial family $(Q_u(y_u))_{u\in U}$. More precisely, we will show that

$$Z^{\#|U\to I|} \in \langle Q_u(y_u)\rangle_{u\in U}\,, \tag{8.1}$$

which means there exists a family $(Z_u)_{u\in U}$ of polynomials of M_C such that

$$Z^{\#|U\to I|} = \sum_{u\in U} Z_u Q_u(y_u).$$

Next, by averaging on all the permutations of the $X_1, \dots, X_m$ we can suppose without loss of generality that Z_u is symmetric for all $u \in U$, so that there exists a lifting $\tilde{Z}_u \in C[\Sigma_1, \dots, \Sigma_n]$ of Z_u for all $u \in U$. Then, by the fundamental theorem of symmetric polynomials, that the lifting is unique, and hence

$$\tilde{Z}^{\#|U\to I|} = \sum_{u\in U} \tilde{Z}_u \tilde{Q}_u(y_u).$$

By evaluating this relation in $(-a_{m-1}, a_{m-2}, \dots, (-1)^{m-1}a_1, (-1)^m a_0)$, we get that

$$z^{\#|U\to I|} = \sum_{u\in U} \tilde{Z}_u \left(-a_{m-1}, a_{m-2}, \dots, (-1)^{m-1}a_1, (-1)^m a_0\right) q_u(y_u).$$

Finally, since each y_u is a root of q_u, we have $z^{\#|U\to I|} = 0$, which means that z is zero.

We prove Formula (8.1) in two steps. We recall that I is the set of pairs $i = (i_1, i_2)$ such that $0 \le i_1 < i_2 \le m$, and that U is a finite subset of R containing $m' + 1$ distinct elements. We define the family $(T_{i,u})_{i\in I, u\in U}$ as the factors of $Q_u(y_u)$. More precisely, for all i in I and u in U, we define

$$T_{i,u} = y_u - X_{i_1} - X_{i_2} - uX_{i_1}X_{i_2} \in M_C \quad \text{so that} \quad \forall u \in U,\ Q_u(y_u) = \prod_{i\in I} T_{i,u}.$$

Now the two steps we prove are

$$\forall f \in U \to I, \quad Z \in \left\langle T_{f(u),u} \right\rangle_{u\in U} \tag{8.2}$$

and

$$\prod_{f\in U\to I} \left\langle T_{f(u),u} \right\rangle_{u\in U} \quad \subset \quad \left\langle \prod_{i\in I} T_{i,u} \right\rangle_{u\in U}. \tag{8.3}$$

By putting the two steps (8.2) and (8.3) together, we get back

$$Z^{\#|U\to I|} \quad \in \quad \prod_{f\in U\to I} \left\langle T_{f(u),u} \right\rangle_{u\in U} \quad \subset \quad \left\langle \prod_{i\in I} T_{i,u} \right\rangle_{u\in U} = \langle Q_u(y_u)\rangle_{u\in U}\,. \tag{8.1}$$

First step : we prove (8.2). Let us take $f \in U \to I$. We want to prove that:

$$Z = \prod_{u<v} P(r_{u,v}) \in \left\langle T_{f(u),u} \right\rangle_{u\in U}$$

Since $\#|U| = m' + 1 > m' = \#|I|$ the function $f \in U \to I$ is non injective, so there exists u and v that have the same image $i = (i_1, i_2)$ by f (this is the pigeon hole principle). Hence both $T_{i,u}$ and $T_{i,v}$ are in $\left\langle T_{f(u),u} \right\rangle_{u\in U}$. But, we have:

$$\begin{cases} T_{i,u} = (r_{u,v} + s_{u,v} - X_{i_1} - X_{i_2}) + u(X_{i_1}X_{i_2} - r_{u,v}s_{u,v}) \\ T_{i,v} = (r_{u,v} + s_{u,v} - X_{i_1} - X_{i_2}) + v(X_{i_1}X_{i_2} - r_{u,v}s_{u,v}) \end{cases}$$

by definition of $T_{i,u}$ and $T_{i,v}$, and because

$$y_u = r_{u,v} + s_{u,v} + ur_{u,v}s_{u,v} \quad \text{and} \quad y_v = r_{u,v} + s_{u,v} + vr_{u,v}s_{u,v}.$$

By inverting this relation, we can get that $r_{u,v} + s_{u,v} - X_{i_1} - X_{i_2}$ and $r_{u,v}s_{u,v} - X_{i_1}X_{i_2}$ are in the ideal $\left\langle T_{f(u),u} \right\rangle_{u\in U}$:

$$\begin{cases} r_{u,v} + s_{u,v} - X_{i_1} - X_{i_2} = & \frac{vT_{i,u} - uT_{i,v}}{v-u} \in \left\langle T_{f(u),u} \right\rangle_{u\in U} \\ r_{u,v}s_{u,v} - X_{i_1}X_{i_2} = & \frac{T_{i,u} - T_{i,v}}{u-v} \in \left\langle T_{f(u),u} \right\rangle_{u\in U}. \end{cases}$$

Now,

$$\begin{aligned} (r_{u,v} - X_{i_1})(r_{u,v} - X_{i_2}) &= r_{u,v}^2 - (X_{i_1} + X_{i_2})r_{u,v} + X_{i_1}X_{i_2} \\ &= (r_{u,v} + r_{u,v} - X_{i_1} - X_{i_2})r_{u,v} - (r_{u,v}r_{u,v} - X_{i_1}X_{i_2}) \\ &\in \left\langle T_{f(u),u} \right\rangle_{u\in U} \end{aligned}$$

So that finally $P(r_{u,v}) \in \left\langle T_{f(u),u} \right\rangle_{u\in U}$, and we can conclude:

$$\prod_{u<v} P(r_{u,v}) \in \left\langle T_{f(u),u} \right\rangle_{u\in U}.$$

Second step: we prove (8.3). This inclusion is completely independent from the definition of $T_{i,u}$ and from the commutative ring M_C it belongs to.

Lemma 8.1. *Given two finite sets I and U, an arbitrary commutative ring A and an arbitrary family $T_{i,u}$ of elements of A, the following inclusion holds:*

$$\prod_{f\in U\to I} \left\langle T_{f(u),u} \right\rangle_{u\in U} \quad \subset \quad \left\langle \prod_{i\in I} T_{i,u} \right\rangle_{u\in U} \tag{8.3}$$



Remark

This Lemma and its proof are actually the only part of this section we formalized in Coq so far.

Proof. Let $(c_{f,u})_{f\in U\to I, u\in U}$ be a family of coefficients in A. We have to prove that

$$\prod_{f\in U\to I}\sum_{u\in U} c_{f,u}T_{f(u),u} \quad \in \quad \left\langle \prod_{i\in I} T_{i,u}\right\rangle_{u\in U},$$

which is equivalent to:

$$\sum_{F\in(U\to I)\to U}\left(\prod_{f\in U\to I} c_{f,F(f)}\prod_{f\in U\to I} T_{f(F(f)),F(f)}\right) \quad \in \quad \left\langle \prod_{i\in I} T_{i,u}\right\rangle_{u\in U}.$$

So, it suffices to prove that for all $F\in(U\to I)\to U$, there exists $u\in U$ such that

$$\prod_{f\in U\to I} T_{f(F(f)),F(f)} \quad \in \quad \left\langle \prod_{i\in I} T_{i,u}\right\rangle_{u\in U}.$$

For this, it suffices to prove that for all F in $(U\to I)\to U$,

$$\exists u\in U,\, \forall i\in I,\, (\exists f\in U\to I, i = f(F(f))\wedge u = F(f)) \tag{8.4}$$

Now, we apply a lemma which is the dual of skolemization and which is valid constructively when stated as follows. We do not give here the proof of this lemma, but we formalized it in a very few lines of SSReflect code.

Lemma 8.2. *Let I and U be two finite sets and let $P(i,u)$ be a decidable property on the elements of these set, then*

$$\exists u\in U,\, \forall i\in I,\, P(i,u) \quad \Leftrightarrow \quad \forall g\in U\to I,\, \exists u\in U,\, P(g(u),u)$$

Using this lemma on the formula (8.4), we get an equivalent formula : for all F in $(U\to I)\to U$,

$$\forall g\in U\to I,\, \exists u\in U,\, (\exists f\in U\to I,\, g(u) = f(F(f))\wedge u = F(f))$$

which has a solution by taking $u = F(f)$ and $f = g$. □

Finally, by putting the two steps (8.2) and (8.3) (with $A = M_C$) together, we get back (8.1) and the proof is complete. □

The idea of this proof is largely inspired by Gauss's proof [38]: we establish an identity between symmetric polynomials that we lift and instantiate in the coefficients of the initial polynomial. However, the identity Gauss established is much more complicated than

$$Z^{\#|U\to I|} = \sum_{u\in U} Z_u Q_u(y_u)$$

The outline of this proof is in fact closer to Laplace proof [61].

Constructive proof, using the Universal Decomposition Algebra

It is possible to add an abstraction layer to the previous proof, using the universal decomposition algebra of $p = X^m + a_{m-1}X^{m-1} + \ldots + a_1X + a_0$ over C. It is defined in [62] as the quotient algebra $A_{C,p} = C[x_1, \ldots, x_n]$ of $M_C = C[X_1, \ldots, X_n]$ by the ideal $\mathcal{I}_p$ generated by symmetric relations.

$$\mathcal{I}_p = \left\langle \sigma_1 + a_{m-1},\, \sigma_2 - a_{m-2},\, \ldots,\, \sigma_{m-1} - (-1)^{m-1}a_1,\, \sigma_m - (-1)^m a_0 \right\rangle.$$

$A = A_{C,p}$ is a C-vector space of dimension $m!$. An important fact is that if u is in C and is zero in A, then it is also zero in C. Also, if $u \in C$ is nilpotent in A, it is zero in C. Now, let us take over the proof at the same point as in Section 8.1, but using the universal decomposition algebra A.

This time, for all u in U, we define

$$q_u = \prod_{i_1<i_2} (X - x_{i_1} - x_{i_2} - ux_{i_1}x_{i_2}) \in A[X].$$

It is symmetric in $x_1, \ldots, x_m$, and thus belongs to $R[X]$, so there exists a root $y_u \in C$ of q_u. But because A is not an integral domain, we do not have necessarily $y_u = x_{i_1} + x_{i_2} + ux_{i_1}x_{i_2}$.

So for all pairs $u, v \in U$ we define again $r_{u,v}$ and $s_{u,v}$ such that

$$r_{u,v} + s_{u,v} + ur_{u,v}s_{u,v} = y_u \quad \text{and } r_{u,v} + s_{u,v} + vr_{u,v}s_{u,v} = y_v,$$

thanks to hypothesis (a.2).

Once again, we pose $z = \prod_{u<v} p(r_{u,v}) \in C$, and this time we use $t_{i,u} = y_u - x_{i_1} - x_{i_2} - ux_{i_1}x_{i_2} \in A$. Now, we show that $z^{\#|U\to I|} = 0$ in A because

$$\forall f \in U \to I, \quad z \in \left\langle t_{f(u),u} \right\rangle_{u\in U},$$

which proof is exactly the same as the one of (8.2), and

$$\prod_{f\in U\to I} \left\langle t_{f(u),u} \right\rangle_{u\in U} \quad \subset \quad \left\langle \prod_{i\in I} t_{i,u} \right\rangle_{u\in U} \quad = \quad \langle q_u(y_u) \rangle_{u\in U} \quad = \quad 0$$

by direct application of Lemma 8.1.

Thus z is also zero in C, and these exist $u, v \in U$ such that $r_{u,v}$ is a root of p in C. □

The universal decomposition algebra plays the role of the splitting field, except it is not an integral domain so that the elements we believe are the roots cannot be directly identified to the proper roots in the universal decomposition algebra.

Abstraction layers and Coq proof

Now is the time to decide at which abstraction level we start the formalization. The very first argument of the induction step of each proof above starts with an argument about the existence of a lifting for symmetric polynomials, so we have to formalize symmetric polynomials and the fundamental theorem of symmetric polynomials anyway.

The second proof would require the formalization of the universal decomposition algebra and its properties as vector space and as an algebra. In the proof of the fundamental theorem of algebra, it abstracts out some uses of the fundamental theorem of symmetric polynomials (mainly the part that uses the uniqueness of the lifting). The formalization of the universal decomposition algebra and its properties could also be useful for other purposes, since it is an object of interest in computer algebra.

However, the use of the universal decomposition algebra does not shorten the proof very much, since only one use of the fundamental theorem of symmetric polynomials would vanish. So we could try to formalize both proofs, beginning with the first one because it requires less prerequisite, and doing the second one afterwards to test the use of the universal decomposition algebra.



Remark

There is another additional abstraction layer we could try to add. We could abstract out the membership of z in the radical ideal

$$\sqrt{\langle q_u(y_u)\rangle_{u\in U}}$$

using Zariski lattices as described in [24]. However, it is unclear how easy it will be to define Zariski lattices and their properties inside Coq.

8.2 Direct construction and other methods

There are other ways to prove that $C = R[i]$ is algebraically closed, some require specific hypotheses, like R being countable starting from (c) instead of (a) and using the decidability of the first-order theory of real closed fields. Let us detail these alternative.

In the same context: the proof we formalized

Another constructive algebraic proof we became aware of after we studied the one presented in Section 8.1 is the one by Derksen [34]. The proof he shows does not rely on an algebraic closure or a splitting field beforehand, it is completely constructive and does not involve symmetric polynomials. It involves a lot of standard results on linear algebra, for which the SSReflect library is adapted. We formalized this proof with no particular difficulty.

When R is countable

It is possible to do a direct construction of the algebraic closure for any countable field. However this does not mean we can isolate its real sub-field and order it (which is a consequence of finding its real part), which is equivalent to constructing a norm operator or a conjugation operator. Since the norm is essential for many developments using complex algebraic numbers, as for example in the proof of the Feit-Thompson Theorem, we still need to work to get it back.

As a consequence it is still useful to define C as $R[i]$ because it is immediately a numeric field, which contains a real sub-field, and for which we can easily define a

norm operator and a conjugation operator. Thus, we still need to prove that $R[i]$ is algebraically closed, but this times the roots can be found in the algebraic closure of C, which make the classical proof constructive in this context.



Remark

When R is the real closure of $\mathbb{Q}$, *i.e.* when R is the field of real algebraic numbers, any polynomial $p \in R[X]$ can be re-expressed as a polynomial $p \in \mathbb{Q}[\alpha][X]$, using the primitive element theorem. But thanks to Kronecker theorem (as described in [63]), we have a factorization algorithm on $\mathbb{Q}[\alpha][X]$ and we can build a splitting field for p step by step.

Using the decidability of the first-order theory of real closed fields

In Part III we show that the theory of real closed fields defined by (c) enjoys quantifier elimination, which entails the decidability of the first-order theory or real closed fields (cf Section 10.4). But the decidability of first-order formulas on R implies the decidability of first-order formulas on C, by a simple encoding of formulas on C to formulas on R (where terms on C are encoded as pairs of two terms on R). Since the formula:

$$\exists x, x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$$

is first order, we can instead suppose

$$\neg \left(\exists x, x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0\right)$$

and try to prove $\bot$. When we try to prove false, we can use the excluded middle, because it is constructively provable that $((A \vee \neg A) \Rightarrow \bot) \Rightarrow \bot$. Using this we could get a splitting field L of p over R, and go on with the classical proof.

Nevertheless, this would prove (c) $\Rightarrow$ (b), which does not fit in the proof of equivalence of (c) $\Leftrightarrow$ (a) $\Leftrightarrow$ (b), and thereby would not provide the implication (a) $\Rightarrow$ (c), which we do not know how to show directly (without going through (b)).



Remark

Both this strategy and the one using Kronecker theorem lead to the construction of a splitting field with an explicit basis, which fits in the framework of the Galois theory developed in SSReflect for the Feit-Thompson Theorem. We could try to formalize a very short proof of the fundamental theorem of algebra which uses Galois theory and the existence of a splitting field with an explicit basis.

Part III

Theory of real and algebraically closed fields

9 Elementary polynomial analysis

This chapter presents the formalization of the elementary theory of roots of polynomials with coefficients in a real closed field. We follow the presentation found in Chapter 2 of [5]. We show however that a formal verification of this chapter imposes some refactoring and reordering. The main issue raised by the formalization of this theory is the formal definition capturing the informal notion of neighborhood. We describe here the solution we have adopted and the alternative proofs we had to design. Of course we do not pretend here to improve the presentation given in [5] which is designed for a human reader. Our version of the proofs might even seem less intuitive or elegant than their paper counterpart. The aim of our description is however to give an insight into the difficulties, or even sometimes the impossibility, of a literal transcription of this chapter of [5] in a machine checked version.

9.1 Direct consequences of the intermediate value theorem

An important consequence of the intermediate value theorem is Rolle's theorem:

```
Lemma rolle : forall a b P, a < b ->
  P.[a] = P.[b] -> {c | c \in `]a, b[ & ((P^`()).[c] = 0)}.
```

where `P^`()` denotes the formal derivative of a polynomial. The proof presented in [5] only describes the case when a and b are "consecutive roots", *i.e.* when P does not vanish on the interval $]a, b[$, and asserts without further comment that this reduction is sufficient to obtain Rolle's theorem. A naive interpretation of this argument would lead to try to establish first that one can obtain the exhaustive list of ordered roots of P and to study the derivative of P between two consecutive points in this list.

Unfortunately, the computation of the list of roots of a polynomial crucially relies on the mean value theorem which in turn is obtained from Rolle's theorem. Basing the proof of Rolle's theorem on the existence of this exhaustive list of roots leads to a circular dependency between Rolle and the mean value theorem. We found out that this untimely use of the exhaustive list of roots can be replaced by a proof by induction. We describe here the sketch of this alternative proof we have formalized.

Alternative proof for Rolle's theorem. We first follow closely the proof in [5] (not

using any induction), but conclude with a weaker statement: at this stage we only show that there is either a root of the derivative or a root of the polynomial itself in the interval, as formalized by:

```
Lemma rolle_weak : forall a b P, a < b -> P.[a] = 0 -> P.[b] = 0 ->
  {c | c \in ']a, b[ & ((P^`()).[c] = 0) || (P.[c] == 0)}.
```

Now we prove Rolle's theorem from this lemma. Let $P \in R[X]$ be a univariate polynomial, and $a, b \in R$ such that $a < b$ and $P(a) = P(b)$. Without loss of generality, we can assume that $P(a) = P(b) = 0$. We reason by induction on the maximal number of roots for the polynomial P in the studied interval. The induction hypothesis is hence:

$$\forall P \in R[X], \forall ab, \quad a < b \ \wedge P(a) = P(b) \ \wedge \ \sharp\{x \mid x \in]a, b[\wedge P(x) = 0\} < n$$

$$\Rightarrow \exists c \in]a, b[, P'(x) = 0$$

for a fixed natural number n. Note that the induction hypothesis applies to any interval, and not only to the one we started with. The base case (for $n = 0$) is trivial because of the strict bound on the number of roots. In the inductive case, we apply the `rolle_weak` lemma to P on the interval $]a, b[$. The conclusion is straightforward in the case the lemma directly provides a root of the derivative. In the other case, the lemma provides a point $c \in]a, b[$ which is not a root of the derivative P' but is a root of the polynomial P. We conclude using the induction hypothesis on the interval $]a, c[$, which contains one root less for P than the initial interval $]a, b[$. □

Once Rolle's theorem is at hand, one can establish the mean value theorem for polynomial functions:

```
Lemma mvt : forall a b P, a < b ->
  {c | c \in ']a, b[ & P.[b] - P.[a] = (P^`()).[c] * (b - a)}.
```

which in turn provides the correspondence between the monotonicity of a polynomial function and the sign of its derivative.

Finally, we recall an important property of polynomials with coefficients in an numeric field. Given an arbitrary non constant polynomial we define its *Cauchy bound* as follows.

```
Definition cauchy_bound (P : {poly R}) :=
  `|lead_coef P|^-1 * \sum_(i < size P) `|P`_i|.
```

which is the sum of the absolute values of the coefficients of the polynomial, divided by the absolute value of its leading coefficient. If a polynomial is nonzero, the absolute value of its roots are bounded by its Cauchy bound:

```
Lemma cauchy_boundP : forall (P : {poly R}) x,
  P != 0 -> P.[x] = 0 -> `| x | <= cauchy_bound P.
```

This result had already been formalized in a previous work [9], following the paper proof presented in [5].

9.2 Root isolation

In our main reference [5], one of the first properties proved in the theory of real closed fields states that if a polynomial does not vanish on an interval, then it has a constant sign on this interval. This is actually a trivial consequence of the intermediate value theorem. The remark following the proof of this property is more problematic: "This proposition shows that it makes sense to talk about the sign of a polynomial to the right (resp. to the left) of any $a \in R$" and this notion of "sign to the right" is used at several places in the sequel of the chapter. Though this makes perfect sense, a constructive formalization of this notion of imposes the computation of the "next root to the right". This definition is left implicit on paper description: readability demands to stay rather vague on the actual value of the bounds of the intervals meeting the requirements the author has in mind. The previously cited remark actually comes as a justification of the lemma explaining the correspondence between the sign of a polynomial P to the right of a point a and the sign of the first derivative of P not vanishing at a. We show in this section that a more precise definition is required in order to prove this lemma, and we describe the solution we have adopted, based on the preliminary formalization of a root isolation process.

Once formalized the results presented in Section 9.1, we can implement and certify the computation of the exhaustive list of ordered roots of a non-zero polynomial P with coefficients in a real closed field.

We fix an arbitrary real closed field `R` and start by defining the following (non Boolean) predicate:

```
Definition roots_on (P : {poly R}) (i : predType R) (s : seq R) :=
  forall x, (x \in i) && (root P x) = (x \in s).
```

The predicate specifies the sequences of elements of `R` which contain all the roots of the polynomial `p` included in the arbitrary subset `i` of the real closed field `R`. It has a small number of useful properties when the set `i` is arbitrary, but we are able to prove a little more results when the set is an interval. For instance one can explain how to concatenate sequences of roots on intervals sharing a bound. Of course the zero polynomial cannot be associated to such a finite sequence on a non-empty interval: hence we show that for any polynomial P and any points a and b, there exists an ordered sequence s such that either P is zero and the sequence is empty, or the sequence contains all the roots of P in the interval $]a, b[$.

Existence of the exhaustive sequence of roots. We fix $P \in R[X]$ be a polynomial and $a, b \in R$. We reason by strong induction on the size of the polynomial P. If $b \leq a$ or if the size (see Section 3.1) of P is zero (which implies that P is zero), then the empty sequence satisfies the requirements. In the inductive case, if the derivative P' is zero, then P is constant and the sequence should be empty. If $P' \neq 0$, the induction hypothesis can be applied to P' and provides the exhaustive sequence of roots of the polynomial P' on the interval $]a, b[$, in order. The rest of the proof consists in studying the interleaving of the roots of P and the roots of P': a root of P' can be a root of P as well, and between two consecutive roots of P', by definition P' has a constant sign, hence P is monotonic and has at most one root. This case study is performed by a nested induction on the sequence of roots of P' obtained from the main induction. □

The algorithm finding the exhaustive list of roots of a polynomial `P` in the interval $]$`a`, `b`$[$ is formalized by the operator:

```
Definition roots (P : {poly R}) (a b : R) : seq R := ...
```

which satisfies the following properties:

```
Lemma roots0 : forall a b, roots 0 a b = [::].

Lemma roots_on_roots : forall P a b, P != 0 ->
  roots_on p `]a, b[ (roots P a b).

Lemma sorted_roots : forall a b P, sorted <%R (roots P a b).

Lemma root_is_roots : forall (P : {poly R}) (a b : R), p != 0 ->
    forall x, x \in `]a, b[ -> (root P) = (x \in roots P a b).
```

In fact, we first build simultaneously the algorithm computing the root isolation and the proof of its specification using a Σ-type.

```
Lemma roots_subproof P a b := {s : seq R | roots_on P `]a, b[ s}.
```

Then the `roots` operator is obtained by projecting this Σ-type on the first, computational component. The atomic specifications above are obtained from the projection of the pair on the second component. The last important property of this ordered sequence of roots is its uniqueness:

```
Lemma roots_on_uniq : forall P a b s1 s2,
  sorted <%R s1 -> sorted <%R s2 ->
  roots_on P `]a, b[ s1 -> roots_on P `]a, b[ s2 -> s1 = s2.
```

Finally, note that to obtain the exhaustive sequence of roots of a polynomial P, it is sufficient to compute this sequence on a sufficiently large interval, for instance $]C(P)-1, C(P)+1[$ where $C(P)$ is the Cauchy bound of the polynomial P (see Section 9.1). We call `rootsR` the function that computes the exhaustive sequence of the roots of a polynomial.

```
Definition rootsR P := roots P (- cauchy_bound P) (cauchy_bound P).
```

9.3 Root neighborhoods

We can now address the formalization of the sign of a polynomial at the right (resp. left) of a given point. This rather informal notion is captured by the sequence of roots we have just defined: the sequence of roots of the derivative of a polynomial gives a precise description of the behavior of a polynomial on an interval since it provides the intervals on which these polynomials have a constant sign. An appropriate and effective definition of neighborhood was actually rather delicate to craft. We start by defining what is the next root of a polynomial after a point `x` and before a point `b`:

```
Definition next_root (P : {poly R}) (x b : R) :=
  if P == 0 then x else head (maxr b x) (roots P x b).
```

where the Boolean expression `(P == 0)` tests whether `P` is the zero polynomial, `maxr` is the binary maximum of two values in the real closed field `R`, and `head` is the head value of a list (with a default value as first argument). The point `(next_root P x b)` is hence equal to:

- `x` if and only if `P` is the zero polynomial or `(b <= x)`
- `b` if `P` has no root in the interval $]$`x`, `b`$[$
- the smallest root of `P` in the interval $]$`x`, `b`$[$ otherwise

It might seem surprising to localize this definition with a right bound: using again the Cauchy bound of the argument `P`, it would be possible to give an absolute definition of the next root for all the points `x` smaller than the biggest root of `P`, and for instance return the Cauchy bound itself for all the points `x` greater that the greatest root of `P`. Another possible default value would be `x` itself in the case of a point on the right of the largest root. But these alternative definitions are in fact soon impractical. Neighborhoods are often used for the study of combinations of polynomials which in general do not share the same Cauchy bound, resulting in unnecessary painful case analysis. More importantly, these two alternative choices introduce spurious side conditions to the algebraic properties we have to establish, like for instance:

```
Lemma next_root_mul : forall (a b : R) (P Q : {poly R}),
  next_root (P * Q) a b = minr (next_root P a b) (next_root Q a b).
```

which expresses that the next root of a product is the minimum of the next roots of each factor. Another possible solution would have been to use an option type but our experience is that the definition we adopted was comfortable enough to spare the burden of handling options. Finally, we define:

```
Definition neighpr (P : {poly R}) (a b : R) := ']a, (next_root P a b)
    [.
```

the neighborhood on the right of the point `a`, on which the polynomial `P` does not change its sign, relatively to the interval $]$`a`, `b`$[$. Similar definitions and properties for left neighborhoods are implemented respectively as `prev_root`, `prev_root_mul` and `neighpl`. These properties of the next (resp. previous) root of a polynomial at a point combine to show that the neighborhood of a product is the intersection of neighborhoods:

```
Lemma neighpl_mul : forall (a b : R) (P Q : {poly R}),
  (neighpl (P * Q) a b) =i [predI (neighpl P a b) & (neighpl Q a b)].
```

where `(_ =i _)` stands for the point-wise equality of the characteristic functions of the intervals and `[predI U & V]` for the intersection of two sets `U` and `V`. Some proofs involving neighborhoods require being able to pick a witness point in the interval they define. This is actually possible in the non degenerate cases:

```
Lemma neighpr_wit : forall (P : {poly R}) (x b : R),
  x < b -> P != 0 -> {y | y \in neighpr P x b}.
```

We now have all the necessary ingredients to formalize the correspondence between the sign of a polynomial p at a point x and the sign at x of the first successive derivative of p which does not cancel:

```
Lemma sgr_neighpr : forall b P x,
  {in neighpr P x b, forall y, sgr P.[y] = sgp_right P x}.
```

This lemma states that on the right neighborhood of a point `x`, the sign of `P` is uniformly given by `(sgp_right P x)`, which computes recursively the first nonzero sign of the derivatives of `P` at `x`, including the 0-th derivative which is `P` itself. It is hence zero only if `x` cancels all the successive derivatives of `P`.

The description of the proof of this property in [5] is a one line remark which recalls that a polynomial P with a root x can be factored by $(X-x)^{\mu(x)}$ where $\mu(x)$ is the multiplicity of x. Although we have defined the multiplicity and proved that this factorization holds, we found that an induction on the size of the polynomial leads to a much more direct proof.

Sign of a polynomial at the right of a point. Let `p` $\in R[X]$ and `x` $\in R$. The proof goes by induction on the size of the polynomial `P`. The base case of a zero polynomial is trivial. In the inductive case, if `x` is not a root of `P` the result is again immediate. Now if `x` is a root of `P`, we denote by s the value of `(sgp_right P x)`, which is by definition the sign at `x` of the first successive derivative of `P` which does not cancel at `x`. Remark that since `x` is a root of `P`, s is also equal to the value of `(sgp_right P^‘() x)`, where again `P^‘()` is the (first) derivative of `P`.

Consider an arbitrary point `y` in the right neighborhood of `x` for `P`. We want to prove that the sign of `P.[y]` is s. Let I be the neighborhood of `x` bounded by `b` for the product of the polynomial `P` by its derivative and `m` be a witness in I. Using the characterization of neighborhood for products of polynomials, we know that `m` belongs to the neighborhood of `x` bounded by `b` for both `P` and its derivative.

Since `y` and `m` are in the same neighborhood for `P`, `P.[y]` and `P.[m]` have the same sign: it is sufficient to prove that the sign of `P.[m]` is s, *i.e.* that the sign of `P.[m]` is `(sgp_right P^‘() x)`.

The left bound of the interval I is `x`, the common left bound of the two intersected neighborhoods. Moreover, by definition of neighborhoods, `P^‘()` has no root in this interval and has hence a constant sign on I. Since `x` is a root of `P`, `P` keeps a constant sign on I, which coincides with the (constant) sign of its first derivative. Hence, since `m` belongs to I, the sign of `P.[m]` and the sign of `P^‘().[m]` are the same. But by induction hypothesis combined with our initial remark, the sign of `P^‘()` on the neighborhood of `x` bounded by `b` for `P^‘()` is equal to s. Since `m` belongs to I, which is itself included in this neighborhood, the sign of `P^‘().[m]` is equal to s. □

The formalization of intervals we described in Section 4.3 plays an important role here to come up with an easy formalization of the easy steps of this proof. The manipulation of neighborhoods and intervals cannot be avoided when proving this lemma formally, whatever version of the proof is chosen. The most pedestrian part of such proofs remains to adjust a neighborhood to make it appropriate for several polynomials. This version of the proof is more friendly than the one based on multiplicities because it limits the number of such explicit computations.

At the time we formalized this, we did not have the “big enough” methodology yet (from Section 5.1.4). We believe that the integration of a variant of this methodology would improve the organization of the code and drastically simplify the proofs involving neighbourhoods.

9.4 About existential formulas

The function `roots` gives the solution to the existential problem:

$$\exists x, (a < x < b) \wedge (P(x) = 0)$$

under the condition that P changes sign between a and b. Thus, the quantifier-free formula $P(a)P(b) < 0$ implies the previous formula. The process of finding a quantifier-free formula equivalent to some quantified formula is called *quantifier elimination*. This small example constitutes the basis of quantifier elimination.

We provide more details in Section 10.3, where we show any problem reduces to the elimination of a single existential quantifier from a formula of the form

$$\exists x, \bigwedge_i P_i(x) = 0 \wedge \bigwedge_j Q_j(x) > 0,$$

where P and Q are two finite families of polynomials. Then, this formula can be re-expressed in an equivalent formula, without quantifier and depending only on the coefficients of the Q_i and P_j. We will exhibit such an expression in Section 11.4.

10

Syntax, semantics and decidability

The first section of this chapter recalls some standard material, essentially following the presentation by W. Hodges [54]. The second section explains the formalization choice we made to represent this standard material within Coq. These are necessary prerequisites for expressing a quantifier elimination function inside Coq.

10.1 First-order logic, the usual presentation

Syntax: Signature, Terms, Formulas

In all what follows, we consider *signatures* of the form: $\Sigma = \mathcal{C} \cup \mathcal{F} \cup \mathcal{R}$, formed of a finite set $\mathcal{C}$ of constant symbols, a finite set $\mathcal{F}$ of function symbols with arity, and a finite set $\mathcal{R}$ of relation symbols with arity. Given such a signature Σ and a countable set of variables $\mathcal{V}$, *terms* are inductively defined as: variables in $\mathcal{V}$ and constants in $\mathcal{C}$ are terms, other terms being of the form $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$ is a function with arity n and $t_1 \ldots t_n$ are terms. A term is *closed* if no variable occur in it. We write $\mathcal{T}(\Sigma, \mathcal{V})$ for terms, and $\mathcal{T}(\Sigma)$ for closed terms.

The *atomic formulas* of a signature Σ are of the form $t_1 = t_2$ where t_1, t_2 are any terms, and $R(t_1, \ldots, t_n)$ where $R \in \mathcal{R}$ is a relation with arity n. The *first-order language* of Σ is the set of all first-order formulas with these atoms. First-order formulas of Σ are recursively defined by: atomic formulas are first-order formulas, other formulas being of the form $\neg f$, $f_1 \wedge f_2$, $f_1 \vee f_2$, $(\exists x, f)$, $(\forall x, f)$, $f_1 \Rightarrow f_2$, where f, f_1, f_2 are formulas.

A formula is *closed* if no variable occurs in it. We write $\mathcal{F}(\Sigma, \mathcal{V})$ for formulas, and $\mathcal{F}(\Sigma)$ for closed formulas.

Any subset of $\mathcal{F}(\Sigma)$ is called a *theory* over Σ. A theory can contain a small set of basic formula which cannot be derived from each other, or an arbitrarily large set of formulas that could all be derived from a smaller set. In this thesis, we call the basic formulas the *axioms*, that the reader must not confuse with *Coq axioms*, as already noted in Section 2.1.

We use the $\vdash$ predicate to denote provability: $T \vdash \psi$ means that ψ is a first-order consequence from formulas in T. The notation $\vec{x}$ will denote a finite list of variables

$x_1, \ldots, x_k$ for some $k \in \mathbb{N}$.

A theory T *admits quantifier elimination* if, for every $\phi(\vec{x}) \in \mathcal{F}(\Sigma, \mathcal{V})$, there exists $\psi(\vec{x}) \in \mathcal{F}(\Sigma, \mathcal{V})$ such that ψ is quantifier-free and

$$T \vdash \forall \vec{x}, ((\phi(\vec{x}) \Rightarrow \psi(\vec{x})) \wedge (\psi(\vec{x}) \Rightarrow \phi(\vec{x})))$$

Semantics: Σ-structures, Models

For any signature $\Sigma = \mathcal{C} \cup \mathcal{F} \cup \mathcal{R}$, a Σ-*structure* is a pair of a set E called the *domain*, and an *interpretation function* I assigning an element of E to each constant symbol in $\mathcal{C}$, a function $E^n \to E$ to each function symbol in $\mathcal{F}$ with arity n, and an n-ary relation on E (*i.e.* a subset of E^n) to each relation symbol in $\mathcal{R}$ with arity n.

For any Σ-structure A, any term $t(\vec{x})$, and any list e of values in the domain of A at least as long as the list of variables $\vec{x}$, we define inductively $[\![\vec{t}(\vec{x})]\!]_{A,e}$ as

- if $\vec{t}(\vec{x})$ is x_i, then $[\![\vec{t}(\vec{x})]\!]_{A,e} = e_i$
- if $\vec{t}(\vec{x})$ is c for some $c \in \mathcal{C}$, then $[\![\vec{t}(\vec{x})]\!]_{A,e} = I(c)$
- if $\vec{t}(\vec{x})$ is $f(\vec{s}(\vec{x}))$ where $f \in \mathcal{F}$, and where $\vec{s}$ are terms with variables $\vec{x}$, then $[\![\vec{t}(\vec{x})]\!]_{A,e} = I(f)([\![\vec{s}(\vec{x})]\!]_{A,e})$

For any Σ-structure A, any atomic formula $\phi(\vec{x}) = R(\vec{t}(\vec{x}))$ where $R \in \mathcal{R}$, and where $\vec{t}$ are terms with variables $\vec{x}$, and any list e of values in the domain of A at least as long as $\vec{x}$, if $[\![\vec{t}(\vec{x})]\!]_{(A,e)}$ is in $I(R)$, we say that A is *a model* of ϕ, denoted by $A, e \models \phi$. This definition is extended to any first-order formula ϕ by induction on the structure of ϕ. We say that a Σ-structure A is a *model* of a theory T, denoted $A \models T$, if and only if $\forall \phi \in T, A \models \phi$.

We say that two formulas $\phi, \psi \in \mathcal{F}(\Sigma, \mathcal{V})$ are T-*equisatisfiable* if in any model M of T, and for any context e, $(M, e \models \phi$ if and only if $M, e \models \psi)$.

We say that a theory T *admits semantic quantifier elimination*, if for every $\phi \in \mathcal{F}(\Sigma)$, there exists $\psi \in \mathcal{F}(\Sigma)$ such that ψ is quantifier free and for any model M of T, and for any list e of values, $M, e \models \phi$ iff $M, e \models \psi$. In this work, we formalize the property of semantic quantifier elimination for the theory of algebraically closed fields.

The theory of algebraically closed fields

The signature of algebraically closed fields is the signature of fields (see Figure 2.3, Section 2.2) $\Sigma_{\text{Fields}} = \{0, 1\} \cup \{-, +, * \cdot^{-1}\} \cup \{\text{unit}\}$. We will also use $\Sigma_{\text{Rings}} = \{0, 1\} \cup \{-, +, *\} \cup \emptyset$. We call *axioms of algebraically closed field*, the set $T_{\text{ClosedFields}}$ of axioms of fields plus an axiom schema $(A_n)_{n \in \mathbb{N}}$ where A_n states the existence of a root for any monic polynomial of degree n:

$$A_n := \forall a_0, \ldots a_{n-1}, \exists x, x^n + a_{n-1}x^{n-1} \cdots + a_1 x + a_0 = 0$$

Theorem 10.1. *$T_{\text{ClosedFields}}$ admits quantifier elimination.*

This result is folklore but it is usually attributed to Tarski [83]. The corresponding geometrical formulation of this result, stating that projections of constructible sets are constructible sets is known as Chevalley's Constructibility theorem [19].

The theory of real closed fields

The signature of real closed fields is the signature of numeric fields (see Section 4.1) $\Sigma_{\text{RealFields}} = \{0,1\} \cup \{-,\ +,\ *\ \cdot^{-1}\} \cup \{\leq, <, \text{unit}\}$. We will also use $\Sigma_{\text{RealDomains}} = \{0,1\} \cup \{-,\ +,\ *\} \cup \emptyset$.



Technical remark

Note that there is no norm $|\cdot|$ symbol in these signature. It is not necessary because it can be encoded by a single (finite) formula using only the symbols from $\Sigma_{\text{RealDomains}}$ and no quantifiers.

We call *axioms of real closed field*, the set T_{RCF} of axioms of numeric fields plus an axiom schema $(\text{IVT}_n)_{n\in\mathbb{N}}$ where IVT_n states the existence of a root for any monic polynomial of degree n that changes sign between two values. Indeed, given a natural number n, we define

$$\text{IVT}_n \ :=\ \forall P_n, \forall x, \forall y,\ x \leq y \ \wedge\ P(x) \leq 0 \leq P(y) \ \Rightarrow\ \exists z,\ x \leq z \leq y \ \wedge\ P(z) = 0,$$

where $\forall P_n$ is an abbreviation for $\forall a_0, \ldots, a_{n-1}$ and $P(x)$ for $a_{n-1}x^{n-1} \cdots + a_1 x + a_0$, so that for a given natural number n, the formula IVT_n is first order.

Theorem 10.2. *T_{RCF} admits quantifier elimination.*

This has been established by Tarski [83] in 1948. This theorem is sometimes also associated with the name of Seidenberg who gave another proof if this result [78] in 1954. The corresponding geometrical formulation of this result is that projections of semi-algebraic sets are semi-algebraic sets [5].

10.2 Formalizing first-order logic in Coq

Our formalization of terms, formulas, models and interpretation is very specific to our context. Although it could be generalized with a model theoretic approach [65], it seems not necessary to treat the specific results of this part. We first present our formalization of models, since it is surprisingly independent from the formalization of terms and formulas. Then we exhibit the encoding of terms and formulas and how they get interpreted into the models.

Models as structures

In our context, we do not formalize model theory. Instead of defining a notion of Σ-structure and model, we prefer to reuse the algebraic hierarchy (see Figure 2.2 and Section 2.2) and the numeric hierarchy (see Figure 4.1 and Section 4.1) as hierarchies of models. Note that the structures that inhabit the interfaces from those hierarchies are always models of the theory described by the signature and the axioms contained in the class of the structure. The theory underling each model is expressed using Coq formulas, and could thus be more expressive than first-order logic. However, one can check that all the axioms that the algebraic and numeric hierarchies contain, are first-order formulas or schemata of first-order formulas.

To highlight the link between models and structures, let us take the example of the `zmodType`, which we recall represents a commutative group. If we flattened the actual content of the $\mathbb{Z}$-module interface (from Section 2.2) we would have the following fields.

```
Structure zmodType := ZmodType{
  M : Type;
  zero : M;
  opp : M -> M;
  add : M -> M -> M;
  _ : associative add;
  _ : commutative add;
  _ : left_id zero add;
  _ : left_inverse zero opp add}.
```

The `zmodType` interface can be seen as the definition of a signature $\Sigma_{\mathbb{Z}\text{modules}} = \{0\} \cup \{-, +\} \cup \emptyset$, together with some axioms $T_{\mathbb{Z}\text{modules}}$, expressed in Coq logic. In this case the axioms are those of the theory of commutative groups, formalized in the interface as unnamed fields. Populating such an interface, *i.e.* building a structure `( Z : zmodType)` is providing a carrier (a type `M`) and interpretations for the symbols, together with a proof that it satisfies the set of axioms $T_{\mathbb{Z}\text{modules}}$, *i.e.* Z is a Σ-structure which satisfies $Z \models T_{\mathbb{Z}\text{modules}}$. So the inhabitants of the interface `zmodType` represent models of $T_{\mathbb{Z}\text{modules}}$. One can see similar correspondences for more complex algebraic interfaces like `ringType`, `unitRingType`, `idomainType`, `fieldType` and other algebraic and numeric interfaces.

We recall that in our setting, the interface `closedFieldType` of algebraically closed field (seen in Section 4.1) is formally defined by packing a structure of field with the following extra axiom schema:

```
Definition closed_field_axiom (R : ringType) : Prop :=
  forall n P, n > 0 ->
    exists x : R, x ^+ n = \sum_(i < n)(P i) * (x ^+ i).
```

where the notation `(x ^+ n)` stands for x^n, and the right hand side of the equation is an iterated sum [8] forming the polynomial expression whose coefficients are given by the `(P : nat -> R)` function.

Similarly, we recall (see Section 4.1.1) that the structure `rcfType` of real closed field is formally defined by packing a structure of field with the following extra axiom schema:

```
Definition real_closed_axiom (R : numDomainType) : Prop :=
  forall (p : {poly R}) (a b : R), a <= b ->
   p.[a] <= 0 <= p.[b] -> exists2 x, a <= x <= b & root p x.
```

We recall that in our context, we restrict our study to the case of *discrete* structures, in particular discrete fields. This means that we assume that there is a Boolean equality test exactly reflecting Coq equality on the terms. For instance a classical formalization of complex numbers could fit this framework through the assumption of a Boolean equality test, and so could a constructive formalization of algebraic numbers.

Terms as inductive types

We reason about first-order logic in Coq, which means we write functions that use terms and formulas as their input or output. To be able to write such functions, we need a deep embedding of first-order formulas in Coq. There are several ways to do so. We describe here the formulation we used.

Terms in $\mathcal{T}(\Sigma_{\text{Fields}}, \mathbb{N})$ are formally described as the inhabitants of the following inductive type:

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

where we reflect division by the product by an inverse. Similarly, first-order formulas in $\mathcal{F}(\Sigma_{\text{Fields}}, \mathbb{N})$ are defined by:

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

where quantifiers explicitly take as argument the natural number representing the name of the variable they bind. And $\mathcal{F}(\Sigma_{\text{RealFields}}, \mathbb{N})$ is defined by

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



Remark

There could be multiple ways to factor `formula` and `oformula`, but we could not decide which one would be the best. In this thesis, we decided to write down the current state of the implementation, despite the redundancy.

We also define notations so that terms and formula are written in a more readable way. For example, the formula:

```
Forall 0 (Exists 1 (Equal (Var 0) (Mul (Var 1) (Var 1))))
```

can be written and displayed:

```
'forall 'X_0, 'exists 'X_1, 'X_0 == 'X_1 * 'X_1
```

in the appropriate scope.

We now define a Coq predicate

```
Definition holds (F : fieldType) : seq F -> formula F -> Prop.
Definition holds_real (F : realFieldType) : seq F -> oformula F -> Prop.
```

such that `(holds F e f)` and `(holds_real F e f)` are both $F, e \models f$ (depending on the signature and the model we use). This requires the definition of the function:

```
Definition eval (F : fieldType), seq F -> term F -> F
```

interpreting terms as elements in the model with respect to a context, such that `(eval F e t)` formalizes $[\![t]\!]_{F,e}$. For instance, the interpretation of the abstract formula:

```
'forall 'X_0, 'forall 'X_1, 'forall 'X_2, 'exists 'X_3,
  'X_0 * 'X_3 * 'X_3 + 'X_1 * 'X_3 + 'X_2 == 0 : formula F,
```

in the environment `[:: a; b; c; x]` is the Coq proposition:

```
forall a b c, exists x, a * x * x + b * x + c = 0 : Prop.
```

For any `(T : Type)`, it is straightforward to test if a formula `(f : formula T)` or `(g : oformula T)` is quantifier-free: we just recursively test that `t` does not feature any `Exists` or `Forall` constructor. This results in two Boolean tests:

```
Definition qf_form (T : Type) : formula T -> bool.
Definition qf_form_real (T : Type) : oformula T -> bool.
```

And when a formula is quantifier-free, it is possible to assign a truth value to it, given a sequence of instantiation for the free variables:

```
Definition qf_eval (F : fieldType), seq F -> formula F -> bool.
Definition qf_eval_real (F : realFieldType), seq F -> oformula F -> bool.
```

Now, the Coq theorems we prove is that there exist two transformations:

```
Definition q_elim_acf (F : closedFieldType) : formula F -> formula F.
Definition q_elim_rcf (F : rcfType) : oformula F -> oformula F.
```

such that:

```
Lemma q_elim_acf_wf (F : closedFieldType) (f : formula F) :
  qf_form (q_elim_acf f).
Lemma q_elim_acfP (F : closedFieldType) (f : formula F) (e : seq F) :
  reflect (holds e f) (qf_eval e (q_elim_rcf f))
```

and

```
Lemma q_elim_rcf_wf (F : rcfType) (f : formula F) :
  qf_form_real (q_elim_rcf f).
Lemma q_elim_rcfP (F : rcfType) (f : formula F) (e : seq F) :
  reflect (holds_real e f) (qf_eval_real e (q_elim_rcf f))
```

These latter theorems are formalizations of the *semantic quantifier elimination*, respectively on discrete algebraically closed fields and discrete real closed fields, assuming that the shallow formalization of models encompasses all models of a given structure.

10.3 Quantifier elimination by projection

For the discrete structures we are interested in, and more generally for first-order theories with decidable atoms, the elimination of a single existential quantifier entails full quantifier elimination. We give here a formal account of this reduction, for the special case of the theory of discrete *rings*. We show also the case of discrete *numeric fields* but we do not detail.

For the theory of rings, we first show that this problem is general enough by describing a transformation `to_rformula` of any formula $f \in \mathcal{F}(\Sigma_{\text{Fields}}, \mathbb{N})$ into a formula $f' \in \mathcal{F}(\Sigma_{\text{Rings}}, \mathbb{N})$ such that f and f' are T_{Fields}-equisatisfiable. We describe the transformation for atoms of $\mathcal{F}(\Sigma_{\text{Fields}}, \mathbb{N})$

- right-hand sides are set at 0 by transforming $(t_1 = t_2)$ into $(t_1 - t_2 = 0)$;
- divisions in the left-hand sides are recursively removed by introducing extra quantifications over fresh variables: $C[t^{-1}] = 0$ is transformed into:

$$\begin{array}{rll} \forall x, & & (x * t - 1 = 0 \wedge t * x - 1 = 0) \\ & \vee & (x = t \wedge \neg\,(\exists x, (x * t - 1 = 0 \wedge t * x - 1 = 0))) \\ \Longrightarrow & & (C[x] = 0)\,. \end{array}$$

- tests of invertibility are removed using the same kind of technique we used for eliminating the inverse. $C[\text{unit } t]$ is transformed into:

$$\begin{array}{rl} & ((\exists x, x * t - 1 = 0 \wedge t * x - 1 = 0) \wedge C[\top]) \\ \vee & (\neg\,(\exists x, x * t - 1 = 0 \wedge t * x - 1 = 0) \wedge C[\bot]) \end{array}$$

This transformation is trivially recursively lifted to any non atomic formula.

For sake of convenience, we introduce a special data-structure for normalized quantifier-free formulas. They can be represented in disjunctive normal form as:

$$\bigvee_{l \in L} \left(\bigwedge_{i \in I_l} t_i = 0 \wedge \bigwedge_{j \in J_l} \neg (t_j = 0) \right)$$

and hence encoded by a list (of sub-formulas in the disjunction), of pairs (one for positive and one for negated atoms) of lists of terms (the left hand sides):

```
seq ((seq term R) * (seq (term R))
```

We consider a discrete field `F`, equipped with an operator:

```
Variable proj : nat -> seq (term F) * seq (term F) -> formula F.
```

whose first integer argument represents the name of a variable, second argument is a quantifier-free conjunctive formula, and which computes a new abstract formula. This operator is meant to eliminate a quantifier from any formula of the form:

$$\exists x_n, \bigwedge_{i \in I} t_i = 0 \wedge \bigwedge_{j \in J} \neg (t_j = 0).$$

We hence require that on a formula on the ring signature, this operator always computes a quantifier-free formula on the ring signature, which we guarantee using the following predicate.

```
Definition wf_proj_axiom := forall i bc,
    dnf_rterm bc -> qf_form (proj i bc) && rformula (proj i bc).
```

We also require this operator must compute a formula equivalent to its input.

```
Definition holds_proj_axiom :=
  forall n bc e,
  let (ex_n_bc := ('exists 'X_n, dnf_to_form [:: bc])%T in
  reflect (holds e ex_n_bc) (qf_eval e (proj n bc)).
```

where `dnf_to_form` converts back the convenient representation we introduced for disjunctive normal form (DNF) quantifier-free formulas to an inhabitant of the type `formula F`.

Under the assumptions that the `proj` operator satisfies the latter two properties `wf_proj_axiom` and `holds_proj_axiom`, we can now prove that the discrete field `F` enjoys full quantifier elimination, meaning that we can implement the `q_elim` function of section 10.3. This quantifier elimination procedure proceeds by recursion on the structure of the formula, eliminating the inner-most quantifier:

- if the formula has the form $\exists x_n F$, where F is quantifier-free, it converts F into DNF. Then since $\exists x, \bigvee (\bigwedge p_i \wedge \bigwedge \neg q_j)$ is equivalent to $\bigvee (\exists x, (\bigwedge p_i \wedge \bigwedge \neg q_j))$, it returns $\bigvee(\text{proj } n\ (p_i)\ (q_j))$.
- if the formula has the form $\forall x_n F$, where F is quantifier-free, it is equivalent to $\neg\exists\neg F$ since the decidability of atoms implies that the full theory is classical. The formula $\neg F$ being quantifier free, the first case method applies to eliminate the quantifier from $\exists\neg F$, and hence from $\neg\exists\neg F$.

The sequential representation of quantifier-free formulas eases the DNF conversions, and their combination with negations in the case of universal quantifiers.

Finally, we obtain a full formal proof that if a discrete field is equipped with a `proj` operator, with a proof of the two `wf_proj_axiom` and `holds_proj_axiom` properties, then we can derive a correct quantifier elimination procedure `q_elim : formula F -> formula F`, which transforms any first-order formula into a quantifier-free one, and such that the input and the output of the quantifier elimination are equisatisfiable in any model of a ring with units.



Remark

For our specific application, `proj` will be implemented using `ProjTermAcf` in Chapter 12.4.

For the theory of numeric domains, all this works exactly the same way, except quantifier-free formulas on $\Sigma_{\text{RealFields}}$ can be represented in disjunctive normal form as:

$$\bigvee_{l\in L}\left(\bigwedge_{i\in I} t_i = 0 \wedge \bigwedge_{j\in J} t_j > 0\right)$$

Thus, let us suppose we have a discrete numeric field equipped with a `proj_real` operator with the following type.

```
Variable proj_real : nat -> seq (term F) * seq (term F) -> oformula F.
```

And let us suppose that this operator satisfies the following two properties.

```
Definition wf_proj_real_axiom := forall i bc, dnf_rterm bc ->
  qf_form_real (proj_real i bc) && orformula (proj_real i bc).
```

```
Definition holds_proj_real_axiom :=
  forall n bc e,
  let (ex_n_bc := ('exists 'X_n, dnf_to_oform [:: bc])%oT in
  reflect (holds_real e ex_n_bc) (qf_eval_real e (proj_real n bc)).
```

where `orformula` and `dnf_to_oform` are the equivalent of respectively `rformula` and `dnf_to_form` but for numeric domains.

Then we can derive a correct quantifier elimination procedure `q_elim_real : oformula F -> oformula F`, which transforms any first order formula into a quantifier-free one, and such that the input and the output of the quantifier elimination are equisatisfiable in any model of a numeric domains with units.

Remark

For our specific application, `proj_real` will be implemented using `ProjTermRcf` in Chapter 12.4.

10.4 Decidability

The first-order theory of a field is *decidable* if one can construct a *Boolean* operator `s : seq R -> formula R -> bool`, which reflects the satisfiability of any formula, *i.e.* satisfies the following property:

```
Definition DecidableField.axiom (s : seq F -> formula F -> bool):=
  forall e f, (holds e f) <-> (s e f = true).
```

This provides a computational characterization of decidability since `s` can be seen as a decision procedure for the first-order theory of `F`.

Of course not all fields have a decidable first-order theory: for instance the field theory of rational numbers is undecidable [76]. However quantifier elimination entails decidability for any first-order theory with decidable atoms. It is hence straightforward to construct by structural recursion a Boolean test `qf_eval` which correctly reflects the validity of such a quantifier-free abstract formula (and remains unspecified on quantified formulas). The correctness of this Boolean test is expressed by the lemma:

```
Lemma qf_evalP : forall (e : seq R) (f : formula R),
  qf_form f -> (holds e f) <-> (qf_eval e f = true).
```

where `qf_form` tests that an abstract formula does not contain any quantifier. The function

```
Definition proj_sat e f := qf_eval e (q_elim f).
```

takes a formula, eliminates its quantifiers, and applies the Boolean satisfiability test `qf_eval` on the result. It is a correct decision procedure as shown by the formal proof that it satisfies the `DecidableField.axiom` specification.

All this works exactly the same way for a decidable first-order theory of numeric fields.

11

Solving polynomial systems of equations (in one variable)

In this chapter we show how to decide whether a system of equations has a solution on a discrete algebraically closed field, and whether a system of inequations has a solution on a discrete real closed field.

More precisely, let C be a discrete algebraically closed field and P and Q be two families of polynomials of $C[X]$, respectively indexed by I and J, two finite subsets of $\mathbb{N}$. We try to tell whether the system of equations

$$\bigwedge_{i\in I} P_i(x) = 0 \wedge \bigwedge_{j\in J} Q_j(x) \neq 0 \text{ with } x \in C \tag{11.1}$$

has a solution or not. We answer this question in Section 11.1.

A more complicated problem is to tell, R being a discrete real closed field and P and Q two families of polynomials of $R[X]$, respectively indexed by I and J, two finite subsets of $\mathbb{N}$, whether the system of inequations

$$\bigwedge_{i\in I} P_i(x) = 0 \wedge \bigwedge_{j\in J} Q_j(x) > 0 \text{ with } x \in R \tag{11.2}$$

has a solution or not.

We find the number of solutions of the latter two systems by programming two counting functions `count_acf` and `count_rcf` which respectively compute the number of solution of the systems (11.1) and (11.2), (respectively with and without multiplicity).

Then, we define two decision procedures `proj_acf` and `proj_rcf` respectively for system (11.1) and system (11.2). Those procedures are obtained by respectively comparing the output of `count_acf` and `count_rcf` to zero. Indeed they return `true` if the corresponding system has at least one solution, and `false` otherwise. As the counting procedures do not take some degenerate cases into account (empty sequences or zero polynomials), each decision procedure is not directly the comparison to zero, but reduces the degenerate problem to a non degenerate one and then compares the count to zero. The procedure `proj_acf` is detailed in Section 11.1.1, while `proj_rcf` is given in Section 11.4.

In Section 11.1, we reduce the problem of deciding whether any of the two systems has solutions to counting the number of roots of a polynomial that satisfy a (possibly empty) list of conditions. We also fully solve the problem for discrete algebraically closed fields (system (11.1)) at this stage, so the following sections only deal with the problem for discrete real closed fields (system (11.2)), which is much more complex.

Then, in Section 11.2 we show that the number of roots of a polynomial satisfying a list of conditions is related to a quantity named Tarski query of two polynomials. In Section 11.3, we relate the Tarski query to another quantity named Cauchy index of a pair of polynomials and we give an algorithm to compute the Cauchy index using coefficients of the polynomials.

This whole part is essentially a reformulation of [5], but reorganized in the way which helped us to formalize it in Coq.

11.1 Reduction of the system

11.1.1 For discrete algebraically closed fields

We want to find the number of solutions (with multplicity) for the system (11.1):

$$\bigwedge_{i\in I} P_i(x) = 0 \wedge \bigwedge_{j\in J} Q_j(x) \neq 0 \text{ with } x \in C \tag{11.1}$$

where C is a discrete algebraically closed field. The problem amounts to finding the number of common roots of the P_i that are not roots of the Q_j. The common roots of the P_i are exactly the roots of $P = \gcd(\{P_i,\ i \in I\})$, the greatest common divisor of the P_i. This first step reduces the problem to finding the roots x of P that satisfy the condition that all the $Q_i(x)$ must be nonzero.

We can go further and form $Q = \prod_{j\in J} Q_j$ which is nonzero exactly where all the Q_j are nonzero. This second step reduces the problem to finding the roots of P that are not roots of Q.

Given two polynomials U and V, we specify $\mathrm{gdco}_U(V)$ to be the greatest polynomial which is coprime with U and a divisor of V. Hence, the roots of $S = \mathrm{gdco}_Q(P)$ are exactly the roots of P that are not roots of Q. Indeed:

1. none of the roots of S are roots of Q, because S is coprime with Q
2. all the roots of S are also roots of P, because S divides P
3. all the roots of P that are not roots of Q are roots of S, because if there exists x such that $P(x) = 0$, $Q(x) \neq 0$ and $S(x) \neq 0$, then $(X - x)S$ would still divide P and be coprime with Q. This is absurd because S was supposed to be maximal.

The computational definition of the *gdco* is obtained by posing $V_0 = V$ and computing $V_{n+1} = \frac{V_n}{\gcd(V_n,U)}$, the *gdco* is V_m for the smallest m such that V_m is coprime with U. Or in Coq:

```
Fixpoint rgdcop_rec U V n :=
  if n is m.+1 then
      if rcoprimep U V then V
      else rgdcop_rec V (rdivp U (rgcdp V U)) m
```

```
  else (U == 0)%:R.

Definition rgdcop U V := rgdcop_rec U V (size V).
```

We prove (lemma rgdcopP in the source code) that this definition corresponds to the specification above.

Technical remark
This algorithm uses a common trick for programming fixpoints when there is no obvious structurally decreasing data. We introduce artificially a natural number n which will be the structurally decreasing argument of the fixpoint and the proof of correction is stated with the invariant hypothesis that n is greater than the size of the second polynomial.

Now, the initial system has as many solutions as $S = \mathrm{gdco}_Q(P)$ has roots. Because C is algebraically closed, the number of roots of S, counted with multiplicity, is given by its degree. So we can pose:

```
Definition count_acf (Ps Qs : seq {poly C}) : nat :=
  let P := \big[@rgcdp _/0]_(p <- Ps) p in (* polynomial P *)
  let Q := \prod_(q <- Qs) q in (* polynomial Q *)
  size (rgdcop P Q) - 1 (* degree of polynomial S, or 0 if S is zero *).
```

which would give the appropriate result provided that at least one polynomial of the sequence `Ps` is non zero. Indeed, if all the polynomials of the sequence `Ps` are zero (or if the sequence is empty), then S is zero and its size is zero which does not correspond to the number of roots of S.

Remark
The *gdco* is designed such that in the degenerate cases, its specification would still hold.

- $\mathrm{gdco}_0(P) = 1$ because there can be no roots of an arbitrary polynomial P that would not be roots of 0.
- When $Q \neq 0$, $\mathrm{gdco}_Q(0) = 0$, because there are only finitely many elements of C that are not roots Q.

Finally, the decision procedure `(proj_acf : seq C -> seq C -> bool)` that determines the satisfiability of the system (11.1) can be defined using `count_acf` by making a particular case when all the `Ps` are zero (because then the result would be `true`).

```
Definition proj_acf (Ps Qs : seq {poly C}) : bool :=
  if all nil Ps then true else count_acf Ps Qs > 0.
```

Remark

In practice, we prefer to redefine it directly in terms of `size`. Indeed (11.1) has a solution if and only S is zero or non constant, which means its size is different from 1.

```
Definition proj_acf (Ps Qs : seq {poly C}) : bool :=
  let P := \big[@rgcdp _/0]_(p <- Ps) p in (* polynomial P *)
  let Q := \prod_(q <- Qs) q in (* polynomial Q *)
  size (rgdcop P Q) != 1 (* polynomial S is non constant or is zero *)
```

11.1.2 For discrete real closed fields

We want to find the number of solutions (without multiplicity) for the system (11.2):

$$\bigwedge_{i\in I} P_i(x) = 0 \wedge \bigwedge_{j\in J} Q_j(x) > 0 \text{ with } x \in R \tag{11.2}$$

where R is a discrete real closed field.

Degenerate case : no P_i

When the list $(P_i)_i$ contains only zero polynomials or is empty, it is not possible to deal as simply as before with the case where all the P_i are zero. Indeed, we can't rely anymore on the zeros of the P_i to provide a finite set for which we could count the number of elements satisfying of the positivity conditions on the Q_j. In order to recover a finite set, we forge the polynomial `bounding_poly`.

Definition: Bounding polynomial

We define the bounding polynomial of a sequence of polynomials $(Q_j)_{j\in J}$ as:

$$(\prod_{j\in J} Q_j)'.$$

If P is the bounding polynomial of the family $(Q_j)_{j\in J}$, we are sure that the signs of the Q_j are all witnessed by $+\infty$, $-\infty$ and the roots of P: between any two sign changes of one of the Q_j there is a root of P. In other words

Lemma 11.1. *If P is the bounding polynomial of the family $(Q_j)_{j\in J}$, then*

$$\exists x, \bigwedge_{j\in J} Q_j(x) > 0 \quad \Leftrightarrow \quad \begin{cases} & \bigwedge_{j\in J} Q_j(+\infty) > 0 \\ \vee & \bigwedge_{j\in J} Q_j(-\infty) > 0 \\ \vee & \exists x, P(x) = 0 \wedge \bigwedge_{j\in J} Q_j(x) > 0. \end{cases}$$

Now we are back to the study of the non degenerate case, using the singleton family P (the bounding polynomial) and the family $(Q_j)_{j\in J}$.

Reduction to one P_i and Q_i

The very first reduction of Section 11.1.1 could be performed the same way. Indeed, when at least one of the P_i is nonzero, the roots x of the P_i such that $\forall j \in J, Q_j(x) > 0$ are exactly the roots y of $P = \gcd(\{P_i,\ i \in I\})$ such that $\forall j \in J, Q_j(y) > 0$.

However, the second reduction of Section 11.1.1, cannot be performed: it is not possible to reduce the system with $(P, (Q_j)_{j\in J})$ to an equivalent system with only one pair P, Q. Nevertheless, `count_rcf` can be computed using several pairs of polynomials $(P, \mathcal{Q}_\alpha)$, where the family $(\mathcal{Q}_\alpha)_{\alpha\in A}$, can be computed using the Q_j. In Section 11.2, we details the relation between the family $(\mathcal{Q}_\alpha)_{\alpha\in A}$ and the family $(Q_j)_{j\in J}$ and we exhibits A.

11.2 Root counting using Tarski queries

For the sake of readability, let us first explain the methodology in the case where the family $(Q_j)_{j\in J}$ is a singleton (Q_1), and let us pose $Q = Q_1$.

Case of one Q

The counting algorithm `count_rcf` should compute the number of roots of P on which Q is positive, or in other words:

$$\sum_{x\in\text{roots}(P)} [\text{sign}(Q(x)) = 1] \qquad \text{where } [\text{false}] = 0 \text{ and } [\text{true}] = 1.$$



Remark

The mathematical notation $[\cdot]$ symbolizes the `nat_of_bool` coercion, which definition is given in Section 1.3. While it is inserted automatically in the Coq code (and not shown it Coq code snippets), we decided to print it out in the mathematical formulas of this thesis for the sake of readability.

Let us generalize this quantity and pose

```
Definition constraints1 (r: seq R) (Q: {poly R}) (s_Q : int) : nat :=
  \sum_(x <- r) (sgr Q.[x] == s_Q).
```

which counts the number of points x from an arbitrary sequence r such that $Q(x)$ has sign $s_Q \in \{-1, 0, 1\}$:

$$\sum_{x\in r} [\text{sign}(Q(x)) = s_Q]\,.$$

This quantity generalizes $\sum_{x\in\text{roots}(P)}[\text{sign}(Q(x)) = 1]$ in two ways: the sign 1 was abstracted by an arbitrary sign s_Q and the list of roots of P by an arbitrary sequence r. So that in the end, computing `(count_rcf P [::Q])` amounts to computing the value of `(constraints1 (roots P) Q 1)`.



Definition: Tarski query

The *Tarski query* of a polynomial Q at a sequence of points r is the sum of the signs taken by Q on the sequence. In other words:

$$\sum_{x\in r} \text{sign}(Q(x)).$$

or in Coq code:

```
Definition taq (r : seq R) (Q : {poly R}) : int :=
  \sum_(x <- r) (sgr Q.[x]).
```

This Tarski query of Q over r is the sum, when x ranges over the sequence of values r, of 1 when $Q(x) > 0$, of 0 when $Q(x) = 0$ and of -1 when $Q(x) < 0$. The signed integer `(taq r Q)` hence gives the number of times $Q(x)$ is positive when x ranges over r, minus the number of time $Q(x)$ is negative when x ranges over this same sequence:

```
taq r Q = \sum_(x <- r) (Q.[x] > 0)
        - \sum_(x <- r) (Q.[x] < 0).
```

This can be rephrased using the definitions we have introduced as:

```
taq r Q = constraints1 r Q 1
        - constraints1 r Q (-1).
```

Moreover, applying the Tarski query to Q^2 and 1, we get more relations between Tarski queries and `(constraints1 r Q sigma)`:

```
taq r (Q ^ 2) = constraints1 r Q 1
              + constraints1 r Q (-1)
taq r 1 = constraints1 r Q 1
        + constraints1 r Q (-1)
        + constraints1 r Q 0.
```

We denote by `(tvec1 r Q)` the row vector gathering the three signed integers `(taq r Q)`, `(taq r (Q ^ 2))` and `(taq r 1)`. We denote by `(cvec1 r Q)` the row vector gathering the three natural numbers `(constraints1 r Q 1)`, `(constraints1 r Q (-1))` and `(constraints1 r Q 0)`. The relations we have stated define a 3×3 linear system:

```
Lemma tvec_cvec1 : forall r Q, tvec1 r Q = cvec1 r Q *m ctmat1.
```

where the 3×3 square matrix `ctmat1` is defined as

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

The determinant of the matrix `ctmat1` is equal to 2, hence we can use its inverse to express `(cvec1 r Q)` in terms of `(tvec1 r Q)`. In particular, the output of `(constraints1 r Q 1)`, which is the first element of the row vector `(cvec1 r Q)`, can be expressed as a linear relation of the Tarski queries of Q, Q^2 and 1. The first column of the inverse of `ctmat1` gives the coefficients of this relation.

Case of multiple Q_j

For the sake of simplicity, let us suppose that the index set J of the $(Q_j)_{j \in J}$ family is $\{1, \ldots, n\}$. In order to generalize to the case where Qs has more than one element, we first generalize the previous `constraints1` operator. We build the generalized version `constraints` of `constraints1` which counts the number of elements of r such that every polynomial in the sequence Qs satisfies the corresponding sign constraint in a sequence of sign constraints $\sigma = (\sigma_1, \ldots, \sigma_n)$.

```
Definition constraints
  (r : seq R) (Qs : seq {poly R}) (sigma : seq int) : nat :=
  \sum_(x <- r) \prod_(i < size Qs) (sgz (Qs`_i).[x] == sigma`_i).
```

Or mathematically:

$$\sum_{x \in r} \prod_{j \in J} [\mathrm{sign}(\mathrm{Q_j(x)}) = \sigma_j].$$

Let us define $A = \{0, 1, 2\}^n$ and $\mathcal{S} = \{-1, 0, 1\}^n$. An element $\alpha \in A$ has the form $(\alpha_1, \ldots, \alpha_n)$ and an element $\sigma \in \mathcal{S}$ has the form $(\sigma_1, \ldots, \sigma_n)$. Moreover, given $\alpha \in A$ we define

$$\mathcal{Q}_\alpha = \prod_{j \in J} Q_j^{\alpha_j}.$$

The case $n = 1$ studied above gives a linear relation between the Tarski queries of a sequence r and the family $(\mathcal{Q}_\alpha)_{\alpha \in \{0,1,2\}}$ (which is exactly $(Q^\alpha)_{\alpha \in \{0,1,2\}}$) and the number of elements x of r that satisfy the constraint $\mathrm{sign}(Q(x)) = \sigma$ for each $\sigma \in \{-1, 0, 1\}$.

We generalize the result for any n by establishing a linear relation between:

- the family `(taq r` $\mathcal{Q}_\alpha$`)` indexed by $\alpha \in A$, where the `taq` operator remains the same as before but is now applied to products of polynomials,
- the family `(constraints r [::Q`$_1$`; Q`$_2$`;...; Q`$_n$`] [::`σ_1`;`σ_2`;...;`σ_n`])` indexed by $\sigma = (\sigma_1, \ldots, \sigma_n) \in \mathcal{S}$.

There are 3^n possible Tarski query expressions, because there is a choice for α_k in three element set of exponents $\{0, 1, 2\}$ for each k in the n element set $\{1, \ldots, n\}$. There are also 3^n for Cauchy index expressions for the exact same reason, except this time it is σ_k that belongs to the three element set of signs $\{1, -1, 0\}$.

Hence, we define `(tvec r Qs)`, the row vector of all possible Tarski query expressions with r and polynomials from `Qs` and `(cvec r Qs)` the row vector of all possible Cauchy index expressions. If we order them properly as shown in [5], we can show that there is a linear system relating the two vectors. More precisely we show that

$$\forall\ \texttt{Qs}, \forall\ \texttt{r}, (\texttt{tvec r Qs}) = (\texttt{cvec r Qs}) \cdot \left(\texttt{ctmat1}^{\otimes(\texttt{size Qs})}\right)$$

where `ctmat1` is the 3×3 square matrix seen above, $\cdot^{\otimes n}$ is the iterated tensor product n times, and `(size Qs)` is the number of elements of `r`. Note that $\texttt{ctmat1}^{\otimes n}$ is still invertible for all n, since the tensor product of two invertible matrices is still invertible. The proof is done by induction over `Qs`.

Yet how to obtain this linear relation is left to the reader in [5] and was a non trivial part of our development. Let us show the outline of this proof.

Proof sketch.

- When `Qs` is the empty sequence `[::]`, the iterated tensor product is the 1×1 square identity matrix and both `(cvec r [::])` and `(tvec r [::])` evaluate to the number of elements of r.
- Otherwise, we prove that

  $$\forall\ \texttt{r}, (\texttt{tvec r (Q :: Qs)}) = (\texttt{cvec r (Q :: Qs)}) \cdot \left(\texttt{ctmat1}^{\otimes(\texttt{size Qs}).+1}\right)$$

  assuming that

  $$\forall\ \texttt{r}, (\texttt{tvec r Qs}) = (\texttt{cvec r Qs}) \cdot \left(\texttt{ctmat1}^{\otimes(\texttt{size Qs})}\right).$$

The proofs is done by expressing `(tvec r (Q :: Qs))` using `(tvec r1 Qs)`, `(tvec r2 Qs)` and `(tvec r0 Qs)`, and also `(cvec r (Q :: Qs))` using `(cvec r1 Qs)`, `(cvec r2 Qs)` and `(cvec r0 Qs)` where

- `r1` is the sub-sequence of `r` where we kept only elements x such that `Q`$(x) > 0$;
- `r2` is the sub-sequence of `r` where we kept only elements x such that `Q`$(x) < 0$;
- `r0` is the sub-sequence of `r` where we kept only elements x such that `Q`$(x) = 0$.

Now we can apply the induction hypothesis on those sequences `r1`, `r2` and `r0`.



Remark

The induction hypothesis applies because we strengthened the induction hypothesis by generalizing it over r. The Proposition 2.68 in Chapter 2 of [5] has a slightly different statement since the Tarski query was not parametrized by a set of elements r but directly by the polynomial P and this makes the generalization over r impossible to do as such. Nevertheless, in the same reference, in Chapter 10, the Proposition 10.62 states the same kind of lemma but with an appropriate definition of Tarski query, with an explicit remark it has been generalized over r (which is named Z in the reference).

□

We do not detail further the formalization of this proof, to the exception of two issues we faced:

- First, we had to take great care about the order in which the coefficients of the `tvec` and `cvec` vectors are given. Fortunately, this task is greatly eased by the system: once programmed an appropriate enumeration of the elements of the vector, the system provides support for the routine bookkeeping.
- The second aspect is the manipulation of matrices defined as dependent types. In the statements above, we have omitted some necessary explicit type casts. Indeed, we compute a row block matrix by gluing three matrices of size `3^n` and we need to get one of size `3^n.+1`. Since `3^n + 3^n + 3^n` and `3^n.+1` are not convertible, the matrix types `'M_(3^n + 3^n + 3^n)` and `'M_(3^n.+1)` are distinct. Therefore, we cannot avoid the use of explicit casts, performed by the following cast operator:

```
Definition castmx : forall (R : Type) (m n m' n' : nat),
  (m = m') * (n = n') -> 'M_(m, n) -> 'M_(m', n') := ...
```

provided by the SSReflect library.

These casts are pervasive in the proofs of the general case, resulting in a considerable amount of uninteresting technical steps in the proofs. On the other hand the design choice for the definition of matrices in the SSReflect library proved very efficient for building a solid corpus of mathematical results. We hope that further evolution of the Coq system, like for instance the "Coq

Modulo Theory" approach [82], will allow for improvement in the manipulations of such datatypes.

Remark

Please remark that we used a 3^n-matrix where the 2^n-matrix $\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}^{\otimes n}$ would have worked. Our main reference on the topic [5] actually needs the 3^n-matrix to refine the relation between the number of roots and the Tarski query. Moreover, it would not have made proofs much shorter nor much simpler to go from a 3^n-matrix to a 2^n.

11.3 Cauchy index

We now define the notion of Cauchy index [18] of the rational fraction $\frac{Q}{P}$ on an interval $]a, b[$. The Cauchy index of the rational fraction $\frac{Q}{P}$ at the point x is defined by:

- -1 if x is a pole and $\lim_{u \to x^-} \frac{Q}{P} = +\infty$ and $\lim_{u \to x^+} \frac{Q}{P} = -\infty$
- 1 if x is a pole and $\lim_{u \to x^-} \frac{Q}{P} = -\infty$ and $\lim_{u \to x^+} \frac{Q}{P} = +\infty$
- 0 otherwise, including when x is not a pole.

Since the Cauchy index of a rational fraction is zero at points which are not poles, this definition can be naturally extended to intervals. The Cauchy index of a rational fraction on an interval $]a, b[$ when a and b are not poles is the sum of the respective Cauchy indexes of the fraction at the poles contained in $]a, b[$, as illustrated on Figure 11.1. The definition also extends to the Cauchy index of a rational fraction on the complete real line $]-\infty, +\infty[$ since the fraction has a finite number of poles. The Cauchy index of a rational fraction at a pole is also called a *jump*. Jumps can

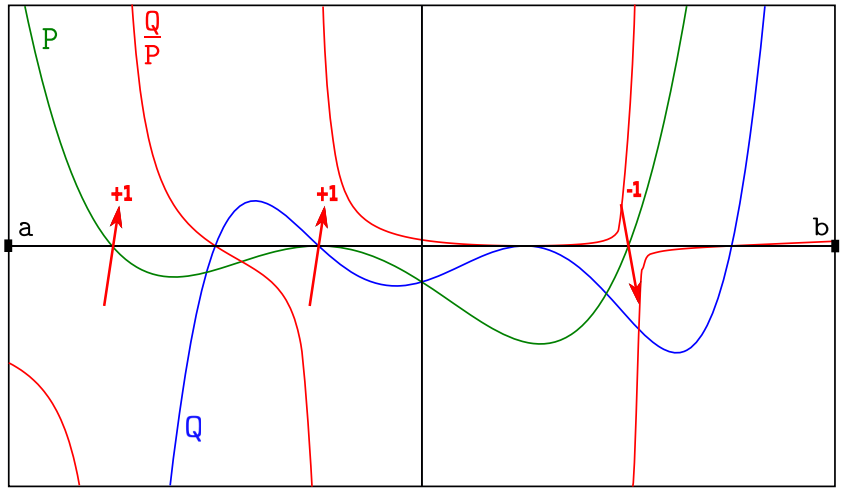


Figure 11.1: Cauchy index on a bounded interval

be defined by replacing the use of limits of rational fractions by considerations on multiplicities. We denote by $\mu_x(P)$ the multiplicity of the point x as root of the polynomial P, this multiplicity is zero if x is not a root of P. Now, $\frac{Q}{P} = (X-x)^{-k}F$ where F is a polynomial fraction that has neither a root nor a pole at x and where $k = \mu_x(P) - \mu_x(Q)$. It is easy to see that $\frac{Q}{P}$ has a zero jump at x if and only if Q is zero or $\mu_x(P)-\mu_x(Q)$ is negative (which means x is not really a pole) or even (which means that the fraction does not change sign during the jump). If $\frac{Q}{P}$ has zero jump at x, the sign of the jump is given by the sign of $\frac{Q}{P}$ at the right of x, which is also the sign of QP at the right of x. These remarks lead to the formalization of `jump` as:

```
Definition jump Q P x: int :=
  let non_null := (Q != 0) && odd (\mu_x P - \mu_x Q) in
  let sign := sgp_right (Q * P) x < 0 in
  ((-1)^+ sign) *+ non_null.
```

which relies again on the coercion `nat_of_bool` (seen in Section 1.3) which interprets `non_null` as the natural number `1` when it is `true` and as `0` if it is `false`. We also benefit from the definition of the sign at the right of a polynomial formalized in Section 9.3. The Cauchy index of a rational fraction $\frac{Q}{P}$ is formalized by summing the values taken by `jump` on the sequence of roots of the denominator P:

```
Definition cindex (a b : R) (Q P : {poly R}) : int :=
  \sum_(x <- roots P a b) jump Q P x.
```

In Figure 11.1, a positive jump is represented by a bottom-up arrow and a negative jump by a top-down arrow.

The key remark is that the value of the jump of $\frac{Q \cdot P'}{P}$ at x is exactly the sign of $Q(x)$, provided that P is nonzero and x is a root of P. But the Cauchy index on a bounded interval sums the jumps of a rational fraction on the sequence of roots of its denominator. So, if we apply `cindex` to $\frac{Q \cdot P'}{P}$, we get:

```
cindex a b (Q * P^‘()) P
  = \sum_(x <- roots P a b) (jump (Q * P^‘()) P x)
  = \sum_(x <- roots P a b) (sgr Q.[x])
```

Now, we remark that the right hand side is equal to the Tarski query of Q on the sequence of roots of P. This leads to the fact that the Cauchy index of $\frac{Q \cdot P'}{P}$ computes the Tarski query of Q on the roots of P in the bounded interval $]a, b[$:

```
Lemma taq_cindex (a b : R) (P Q : {poly R}) :
   taq (roots P a b) Q = cindex a b (P^‘() * Q) P.
```

Since we are interested in all the roots of P, we pose:

```
Definition cindexR Q P := \sum_(x <- rootsR P) jump Q P x.
```

We recall that the list `(rootsR p)` gathers all the roots of P, since there are no roots outside of `‘](- cauchy_bound P), (cauchy_bound P)[`, according to the properties of the Cauchy bounds. We also defined a version of the Cauchy index to take every pole of the fraction into account.

Now the Cauchy index on R can be expressed in term of variation of signs of pseudo remainder sequences, which is a purely algebraic expression involving the coefficients of `P` and `Q`. The pseudo remainder sequence of two polynomials is defined

as the sequence whose first two elements are the polynomials given as arguments and recursively defined by appending the pseudo-remainder of the last two polynomials of the sequence.

```
Definition next_mod P Q := -(lead_coef Q ^+ rscalp P Q) *: rmodp P Q.

Definition mods P Q :=
  let fix aux P Q n :=
    if n is m.+1
      then if P == 0 then [::] else P :: (aux Q (next_mod P Q) m)
      else [::] in aux P Q (maxn (size P) (size Q).+1).
```

When we count the number of sign changes of the sequence `(mods P Q)` evaluated at a point `a` and we subtract the number of sign changes of `(mods P Q)` evaluated at a point `b`, we get back `(cindex a b Q P)`. Let us detail this.

More formally, the number of sign changes of a list of values can be defined using the `changes` function.

```
Fixpoint changes (s : seq R) : nat :=
  if s is x :: Q then (x * head 0 Q < 0) + changes Q else 0.
```

Example

For example, `changes` $[1,-1]$ is 1, `changes` $[1,-1,1]$ is 2 and `changes` $[1,-1,1,1]$ is 2

Remark

We do not give examples using zero because the sign zero will never be contained in a sequence we give to `changes`, throughout this work.

Then we define a function `changes_horner` to evaluate an arbitrary list of polynomials `sP` in a point `x` and take the sign changes of this list.

```
Definition changes_horner (sP : seq {poly R}) x :=
  changes (map (fun P => P.[x]) sP).
```

And we subtract two number of changes in two different points, using

```
Definition changes_itv_poly a b (sP : seq {poly R}) :=
  changes_horner sP a - changes_horner sP b.
```

And we can prove formally that for two polynomials P and Q, as soon as a and b are not roots of any polynomial occurring in `(mods P Q)`, the Cauchy index of Q/P on $]a,b[$ coincides with the difference of number of sign changes between a and b in their pseudo remainder sequence, *i.e.* that

```
changes_itv_mods a b P Q = cindex a b Q P.
```

However, we are interested in the sign changes between $-\infty$ and $+\infty$ instead of between a and b, and rather than using the Cauchy bounds of all the polynomials in `(mods P Q)`, it is easier to find the sign of a polynomial in infinite points by looking at its leading coefficient:

```
Definition sgp_pinfty P := sgr (lead_coef P).
Definition sgp_minfty P := sgr ((-1) ^+ (size P).-1 * (lead_coef P)).
```

And we can define the sign changes of an arbitrary sequence of polynomials `sP` in infinite points, subtract them and apply the algorithm to the pseudo-remainder sequence `(mods P Q)`.

```
Definition changes_pinfty sP := changes (map lead_coef sP).
Definition changes_minfty sP :=
  changes (map (fun P => (-1) ^+ (~~ odd (size P)) * lead_coef P) sP).

Definition changes_poly (sP : seq {poly R}) :=
  changes_minfty sP - changes_pinfty sP.
Definition changes_mods P Q := changes_poly (mods P Q).
```

And similarly we can prove formally that given two polynomials P and Q, the Cauchy index of Q/P on $]-\infty,+\infty[$ is exactly the difference of the number of sign changes of their pseudo-remainder sequence between $-\infty$ and $+\infty$:

```
Lemma changes_mods_cindex P Q : changes_mods P Q = cindexR Q P.
```

Following the presentation of [5], the proof of this lemma goes by induction on the length of the sequence of pseudo-remainders, relying on the fact that `cindexR` and `changes_mod` satisfy the same recursion relation:

```
Lemma cindexR_rec P Q :
  cindexR Q P = crossR (P * Q) + cindexR (next_mod P Q) Q.
Lemma changes_mods_rec P Q :
  changes_mods P Q = crossR (P * Q) + changes_mods Q (next_mod P Q).
```

where `(crossR P)` is defined as `1` if `P` is non-positive in $-\infty$ and non-negative in $+\infty$, `-1` if it is non-negative in $-\infty$ and non-positive in $+\infty$ and `0` otherwise. More formally, we define a couple of very simple notions:

```
Definition variation (x y : R) : int := (sgz y) * (x * y < 0).
Definition crossR P    := variation (sgp_minfty P) (sgp_pinfty P).
Definition cross P a b := variation P.[a]      P.[b].
```

The function `variation` applied in two numbers x and y gives 1 if none of them are zero and if $x < y$, it gives -1 if none of them are zero and if $y < x$, and it gives 0 otherwise. Now, the function `crossR` gives the variation of a polynomial between $-\infty$ and $+\infty$ and `cross` between two arbitrary numbers. These very simple notions help to break down the definitions of jumps and sign changes of a polynomial into smaller pieces.

Detailing why the relation `cindexR_rec` holds was however far more technical to conduct than suggested by the reference [5]. As described in the reference, it is done in two steps.

```
(* first  step *)
Lemma cindex_mod a b P Q : cindex a b P Q =
   (sgz (lead_coef Q) ^+ rscalp P Q) * cindex a b (rmodp P Q) Q.

(* second step *)
Lemma cindex_inv a b : a < b -> forall P Q,
  ~~ root (P * Q) a -> ~~ root (P * Q) b ->
  cindex a b Q P + cindex a b P Q = cross (P * Q) a b.
```

The first step is quite easy to show. However, the second step is non trivial and involves a proof about summing crosses along a sequence. Indeed, the sum of `(cindex a b Q P)` and `(cindex a b P Q)` is the sum of contiguous jumps, which is equal to the sum of crosses between each next number of the sequence beginning with `a`, ending with `b` and containing in between the ordered sequence of mid points of `(roots a b (P * Q))`. Now, the sum of variations along a sequence is the variation between the first and the last point of the sequence, which means that the sum of all the crosses of PQ can only be the cross of PQ between $-\infty$ and $+\infty$.



Remark

In the end, what was non trivial was to find the appropriate notions (`variation`, `cross`, `sgp_right` and neighborhoods). In the last iteration of the proof, this lemma was not difficult to prove at all. Indeed, as soon as we introduced the simple concepts of `variation` and `cross`, despite their simplicity, it caused a major refactoring in the proof, diminishing its size by approximately one half. Moreover, before the refactoring, there was a difficult equality to prove, involving ring additions and multiplications. It was done (painfully) by hand, as the decision procedure for ring expressions equality was not compatible with SSReflect rings, (a good interfacing with the `ring` tactic would have solved the equality). However, the introduction of `variation` and `cross` caused this equation to vanish, as it was certainly abstracting it out in terms of a combination of elementary properties of `variation` and `cross`. This was very surprising and satisfactory.

11.4 Algebraic formula characterizing the existence of a root

The number of roots of the P_i which makes the Q_j positive is given by the function `count_rcf` defined by the following algebraic expression.

$$\left(\sum_{\alpha\in\{0,1,2\}^n} \lambda_\alpha \cdot \left(\texttt{var_mods_infty}\quad P\left(P'\cdot\prod_{k\in\{1,\ldots,n\}} Q_k^{\alpha_k}\right)\right)\right)$$

provided that P is nonzero.

This expression is obtained by replacing `constraints` by the corresponding expression in terms of `taq`, then the `taq` by the appropriate `cindex` (given by the lemma `taq_cindex`) and finally replacing `cindex` by `changes_mods` (given by the lemma `changes_mods_cindex`).

Now we take into account the study of the degenerate case (when all the P_i are null) described in Section 11.1.2. Hence, the procedure `proj_rcf` tests if $\gcd((P_i)_i)$ is zero, if it is not zero then it uses directly `count_rcf`, otherwise it uses the equivalent formula provided by Lemma 11.1 and eliminates using `count_rcf` the existential quantifier in the third member of the disjunction.

The procedure `proj_rcf` is defined by:

```
Definition proj_rcf (sP sQ : seq {poly R}) :=
  let P := \big[rgcdp/0]_(P <- sP) P in
  if P != 0
    then (* non degenerate case -> direct use of count_rcf *)
      0 < count_rcf P sQ
    else (* degenerate case, left hand side of Lemma 11.1 *)
      let P := bounding_poly sQ in
      [|| \big[andb/true]_(Q <- sQ) (sgp_pinfty Q > 0)
        , \big[andb/true]_(Q <- sQ) (sgp_minfty Q > 0)
        |  0 < count_rcf P sQ].
```

The procedure `proj_rcf` only involves comparisons between polynomial expressions in the coefficients of the polynomials featured by the atoms of the initial formula. Though this final expression may be unreadable by human eyes as such, programming this combination of all the elementary steps presented in this section raises no particular difficulty.

12

Programming and certifying the quantifier elimination

We now describe how the results of the previous chapters provide a full quantifier elimination algorithm. Until now, we formalized the theory of Coq univariate polynomials with coefficients in the model. What we need to do is to treat the theory of formal multivariate polynomials. We need to transform some programs that manipulate Coq polynomials into programs that deal with formal multivariate polynomials. While studying the theory of real closed fields and while the proof of correction of `proj_rcf` was quite long, only a small part describes the computational content required: the functions that were used to code `proj_rcf`. And only this part of the code may be reprogrammed and proved correct for formal multivariate polynomials.

We first show the expected behaviour of our code on a simple example.

12.1 An example

Let us eliminate quantifiers on the formula

$$\exists x, (ax^2 + bx + c = 0) \wedge (x > 0),$$

where x is the variable to eliminate and a, b and c are parameters: free variables which the resulting formula may mention.

In the `proj_rcf` procedure described in Section 11.4, we perform pseudo-divisions, and thus we have to test whether some coefficients are null or not. Moreover, when we compute the number of sign changes of the pseudo remainder sequence, we test the sign of some coefficients. In the formal multivariate case, those coefficients may contain formal variables. The result depends on the possible instantiation of those variables.

For example, let us pose $P(x) = ax^2 + bx + c$ and $Q(x) = x$. According to `proj_rcf`, we first compute $P'Q = 2ax^2 + bx$. Then we study the pseudo remainder sequence of P by $P'Q$, which is $ax^2 + bx + c$, $2ax^2 + bx$, $-2a^2bx - 4a^2c$ and $32a^8b^2c(b^2 - 4ac)$, provided that none of the leading coefficients of those polynomials are zero.

If $a = 0$, the sequence becomes: $bx + c$, bx, $-b^2c$, provided that b and c are nonzero. If moreover $b = 0$, the sequence is c and if c is also zero, the sequence is empty.

Let us go back to the case where $a \neq 0$. Maybe $2a^2b = 0$, in which case the sequence would be: $ax^2 + c$, $2ax^2$ and $4a^2c$. And if $4a^2c = 0$, then it would be: ax^2 and $2ax^2$.

Finally, if $a \neq 0$ and $2a^2b \neq 0$, maybe $32a^8b^2c(b^2 - 4ac) = 0$ and the sequence would be: $ax^2 + bx + c$, $2ax^2 + bx$, $-2a^2bx - 4a^2c$

In fact, taking the pseudo-remainder sequences of polynomials with parameter makes no sense if the values of the parameters are discarded. To take them into account, we must include the possible conditions along with the result in each branch.

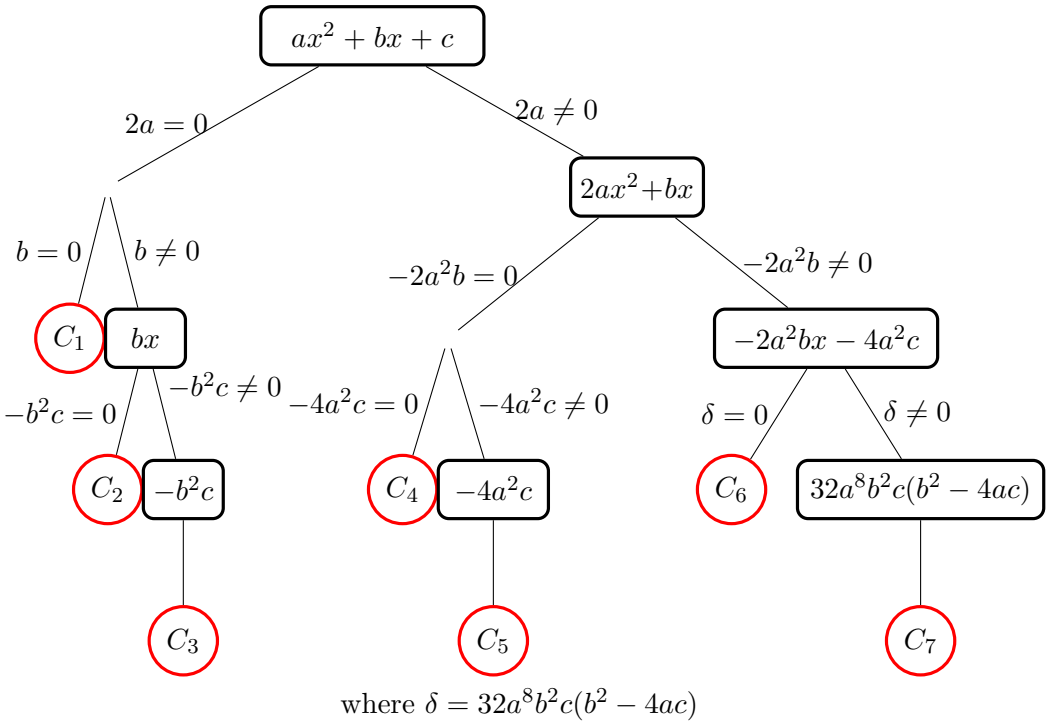


Figure 12.1: Pseudo remainder tree

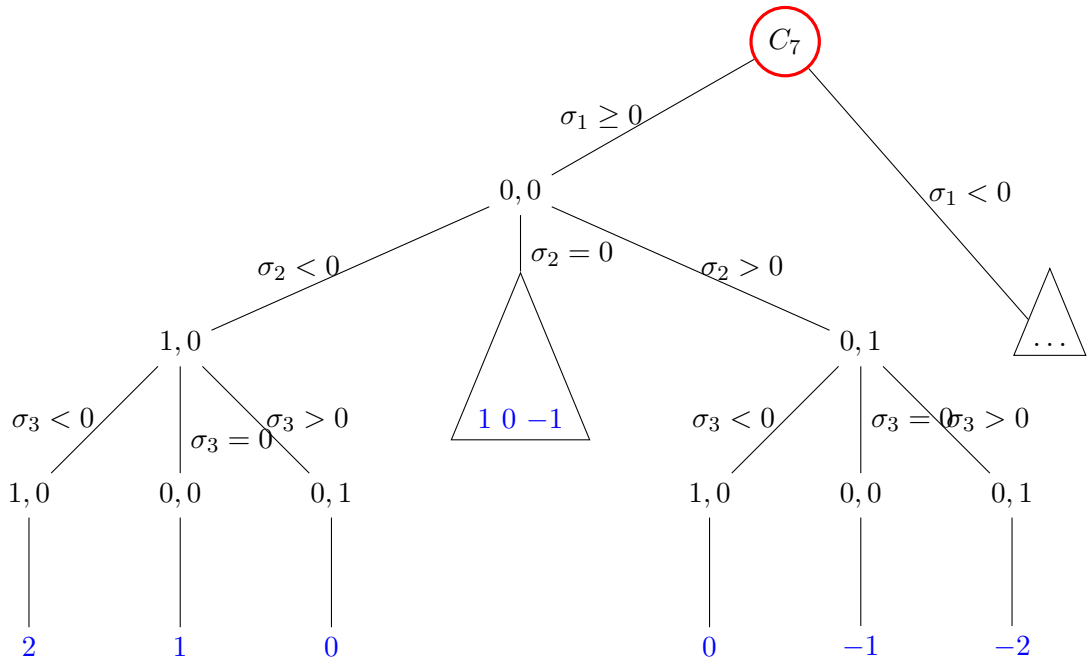
The figure 12.1 represents the tree of possible pseudo-remainder sequences for our example. The branching is based on the nullity or non nullity of the leading coefficients of the next polynomial in the sequence. The right-most branch of this tree is the pseudo-remainder sequence we obtain in the non degenerate case, while each left-branching corresponds to a degeneration.

Each leaf can be labelled with the conjunction C_i of all the conditions one can find along the branch. For example,

$$C_5 = (2a \neq 0) \wedge (-2a^2b = 0) \wedge (-4a^2c \neq 0)$$

These conjunctive formulas represent the equations of regions of the space of parameters on which the pseudo-remainder sequence described by the branch is valid. We call such regions cells. For example, on the region described by C_5, the pseudo remainder sequence of $P = ax^2 + bx + c$ and $P'Q = 2ax^2 + bx$ is

$$\left[ax^2 + bx + c,\ 2ax^2 + bx,\ -4a^2c\right]$$



where $\sigma_1 = a * 2a$, $\sigma_2 = 2a * -2a^2b$ and $\sigma_3 = -2a^2b * \delta$

Figure 12.2: Differences of sign changes on C_7

Then, we have to compute the sign changes of the possible pseudo remainder sequences in $-\infty$ and $+\infty$. Again, the result depends on the values of the parameters a, b and c, but this time, it triggers ordering conditions on the parameters.

Figure 12.2 shows the possible outcomes in the computation of the Tarski query of P and Q, depending on sign tests on the coefficients of the polynomials involved in the sequence, which themselves depend on the cell we are in (in this example C_7). The labels on the edges correspond to the signs we have to test for, the node labels of the form i, j correspond respectively to the number of sign changes in $-\infty$ and in $+\infty$, and the numbers at the leafs is the difference between the sum of the sign changes in $-\infty$ and in $+\infty$.

In order to get the quantifier-free formula equivalent to

$$\exists x, \quad ax^2 + bx + c = 0 \wedge x > 0$$

we would also have to perform the same tasks we did but with P, Q^2 and $P, 1$ instead of P, Q. With the values of the Tarski queries of both P, Q and P, Q^2, we can compute `(count_alg [::P] [::Q])`, the number of roots of P that make Q positive, for each cell. Then, the resulting quantifier-free formula is the disjunction over all the cells C of the conjunction of the formula that state that the count is positive on C the and the formula characterizing the cell C.

In other words, if $\mathcal{C}$ is the set of all the formulas characterizing the cells that were found in the process of the computation of `(count_alg [::P] [::Q])`, and if F_C for each $C \in \mathcal{C}$ is the formula that states `(count_alg [::P] [::Q] > 0)` on C, the resulting quantifier-free formula is

$$\bigvee_{C \in \mathcal{C}} (C \wedge F_C).$$

If we computed, factorized and simplified a bit this formula, we would get a formula like:

$$\begin{array}{rcccc} & a=0 & \wedge & (b=0 & \wedge & c=0) \\ & & \vee & (b\neq 0 & \wedge & bc<0) \\ \vee & a\neq 0 & \wedge & (\Delta>0 & \wedge & b^3>a^2b\Delta) \end{array}$$

where $\Delta = b^2 - 4ac$.

Remark

We remark that conditions on the discriminant $b^2 - 4ac$ naturally appeared in the process, which were indeed the necessary condition to decide whether an arbitrary second degree polynomial has a real root or not.

Remark

In order to keep this example to reasonable size, we simplified the formulas and branching to the extreme. The procedure we implemented fails way before providing the pseudo remainder tree given in Figure 12.1. The is due not only to the natural complexity of the procedure, but also to our use of naive algorithms and structures. Indeed, we did not even try to factorize the uses of equality or sign tests.

12.2 Algorithm transformation and projection

For the sake of readability, we now confuse `formula` and `oformula` and always write `formula`. We let the reader interpret correctly which formula it is, depending on the context: for field formulas it is `formula` and for numeric field formulas it is `oformula`. Often, we will also drop the argument `F` of `term` and `formula` when there is no ambiguity.

Instead of building a tree like in Section 12.1 or like in our main reference on this subject [5], we show how to systematically transform functions operating on the type `F`, `{poly F}` and `bool` to functions operating on their respective reifications `(term F)`, `polyF` and `formula F`, where `polyF` is the type of polynomials of terms, simply defined as sequences of terms:

```
Definition polyF := seq (term F).
```

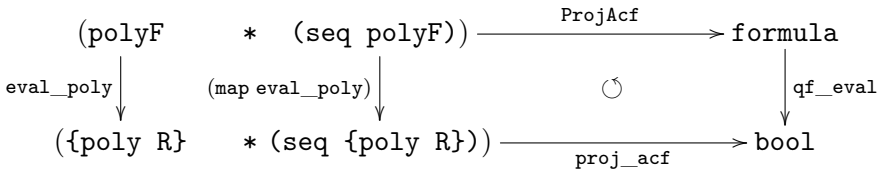
We also define the evaluation function `eval_poly` on reified polynomials by a simple mapping of `eval` on each coefficient of the reified polynomial.

On one hand we call `F`, `{poly F}` and `bool` *reifiable types* and on the other hand we call `(term F)`, `polyF` and `(formula F)` their *reified counterparts*.

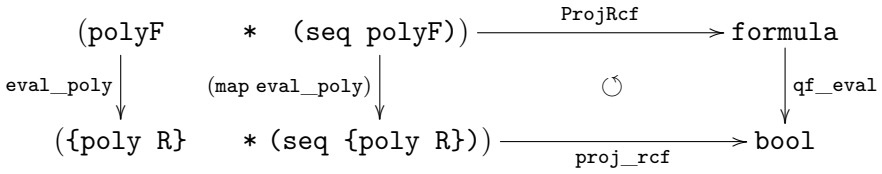
In Section 12.4, we lift `proj_acf` and `proj_rcf` to their reified form, which lead to the following functions.

```
Definition ProjAcf : seq polyF -> seq polyF -> formula.
Definition ProjRcf : seq polyF -> seq polyF -> formula.
```

Those functions are such that the following diagrams commute.



The arguments of the functions `ProjAcf` and `proj_acf` are on the left hand side of the diagram. We represent them in a non-curried style on the diagrams.



The arguments of the functions `ProjRcf` and `proj_rcf` are on the left hand side of the diagram. We represented them in a non-curried style on the diagrams.

Now, quantifier elimination by projection (cf Section 10.3) expects the following functions.

```
Definition ProjTermAcf :
  nat -> seq (term F) * seq (term F) -> formula F.
Definition ProjTermRcf :
  nat -> seq (term F) * seq (term F) -> formula F.
```

Those functions can be defined from `ProjAcf` and `ProjRcf` by first abstracting the two sequence of terms to two sequences of polynomials, according to the variable number given to the function. This is done through the function `abstrX`.

```
Definition abstrX (i : nat) (tf : term F) : polyF.
```

12.3 Direct transformation

Given a function g with inputs and outputs in reifiable types (i.e `F`, `{poly F}` and `bool`), we call it a DT-function if we can find a function `G`, with the same signature except all the types are replaced by their reified counterpart.

Definition: DT-function and direct counterpart

Given $n+1$ types $A_1, \ldots, A_n, B$ and their formal counterparts $A'_1, \ldots, A'_n, B'$, a function $g : A_1 \to \ldots A_n \to B$ is a DT-function (Directly Transformable function) if we can program its reified counterpart

$$\texttt{G} : A'_1 \to \ldots \to A'_n \to B'$$

such that the following evaluation/interpretation diagram commutes:

$$\begin{array}{ccc} (A'_1 * \ldots * A'_n) & \xrightarrow{\texttt{G}} & B' \\ \Big\downarrow{\scriptstyle \texttt{eval}} & \circlearrowleft & \Big\downarrow{\scriptstyle \texttt{eval}} \\ (A_1 * \ldots * A_n) & \xrightarrow[g]{} & B \end{array}$$

Moreover, `G` is called the direct (reified) counterpart of g.

Remark

Now, the goal of this section amounts to prove that both `proj_acf` and `proj_rcf` are DT-functions.

Technical remark

The definitions and theorems we give in this section have not been formalized in Coq: they are meta-theorems which gave us guidelines for this formalization. However, we believe that with some effort, it is possible to formalize them in Coq, anyway. This would involve the deep embedding of a programming language in Coq in order to generate both functions acting on `F`, `{poly F}` and `bool`, and functions acting on their reified counterparts.

Examples of DT-functions are arithmetic operations on terms (`Add`, `Opp`, `Mul`, ...) and on polynomials (`AddPoly`, `OppPoly`, `MulPoly`, ...). Since the method to get the direct counterpart of a DT-function is generic, we here show it on some examples of DT-functions for the sake of simplicity.

To turn a DT-function operating on values in the real closed field and on polynomials into its reified counterpart, we examine its code and turn each instruction into its formal counterpart. For example, the function `(fun x : R => x * x)` that computes the square of an element of `R` is turned into `(fun x : term => Mul x x)`, which returns a `term`. The function `(fun x : R => x < 1)` that tests whether an element of `R` is greater that `1` is turned into `(fun x : term => Lt x 1)` which returns a formula. Indeed, their evaluation/interpretation diagrams commute trivially.

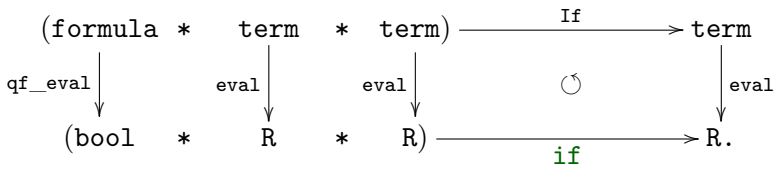
All but one of the transformations are straightforward. Let us consider for example the function `lead_coef` that returns the leading coefficient of a polynomial:

```
Fixpoint lead_coef (p : {poly R}) : R :=
  match p with
  | [::] => 0
  | a :: q => if q == 0 then a else lead_coef q
  end.
```

Now let us try to turn it into its formal counterpart `LeadCoef`. The destruct construction `(match _ with _ end)` is the same in both procedures (because of the

encoding of both polynomials representation are the same). However the conditional (`if q == 0 then _ else _`) cannot be translated directly. Indeed one cannot know whether a formal value is null without knowing the values taken by the free variables. As a consequence we cannot determine which branch of the conditional to take: the formula has to collect all cases and link the values taken by the conditional expression with the conditions discriminating the different branches.

We can see the `if` construction as a function taking three arguments — a condition an two expressions of some type — and returning a value of the same type. Because evaluation functions take environments (i.e. values to give to the free variables) as arguments, there is no way to find a function `If` such that the following evaluation/interpretation diagram commutes:



As a consequence, we could not find a formal counterpart of `lead_coef` with type `polyF -> term`. This means that we could not prove `if` and `lead_coef` were DT-functions. More generally, there is no direct way to find a formal counterpart to the code of an arbitrary function g that uses non DT-functions. However, it is important to notice that even if the code of a function g cannot be translated directly, it might still be a DT-function. In particular, `proj_acf` and `proj_rcf` cannot avoid using conditional statements, but in the end we will still show they are DT-functions.

12.4 Continuation passing style transformation

To find some reified counterparts to non DT-functions, we introduce a different reified formal counterpart to the `if` construct. We call it its cps-counterpart, for continuation passing style counterpart.

The cps-counterpart to the `if` is defined as :

```
Definition If (cond th el : formula) : formula :=
  Or (And cond th) (And (Not cond) el).
```

This formula requires `th` to be satisfied when `cond` is and `el` to be satisfied when `cond` is not.

With this definition, `If` does not take an arbitrary type for its arguments anymore, but only formulas. Hence any function which uses a conditional statement must then output a formula, which is fair in our setting since we are ultimately interested in building the `ProjRcf` and `ProjAcf` functions, which outputs a formula.

We propose the following cps-transformation for the function `lead_coef`:

```
Fixpoint LeadCoef (P : polyF) (k : term -> formula) : formula :=
  match p with
  | [::] => k 0
  | a :: q => If (q == 0) (k a) (LeadCoef q k)
  end.
```

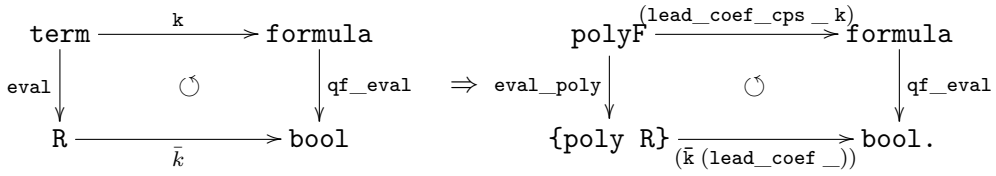
where the additional argument `k` is called a continuation.

The correctness of `LeadCoef` with regard to `lead_coef` is expressed by the following lemma.

```
Lemma LeadCoefP : forall (k : term -> formula) (k̄ : R -> bool),
  (forall x e, qf_eval e (k x) = k̄ (eval e x))) ->
  forall P e, qf_eval e (LeadCoef P k) = k̄ (lead_coef (eval_poly e P)).
```

where `(k̄ : R -> bool)` is the interpretation of `(k : term -> formula)`. This lemma expresses that executing `LeadCoef` on a polynomial `P` with continuation `k` and interpreting the result in environment `e` leads to the same result as executing `lead_coef` on the interpretation of the polynomial `P` and then applying the continuation. The hypothesis of this lemma says that the continuation must commute with evaluation.

This can be expressed by the following implication of the evaluation/interpretation diagram, which correspond to composition of `lead_coef` and `k`.



More generally, we have the following theorem.

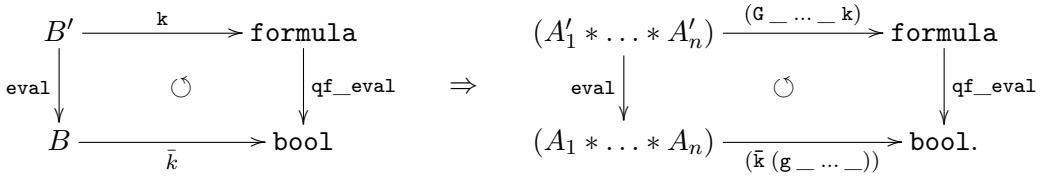
Theorem 12.1 (meta-theorem on cps-counterpart)**.** *Given $n+1$ types $A_1, \ldots, A_n, B$ and their formal counterparts $A'_1, \ldots, A'_n, B'$, any function*

$$g : A_1 \to \ldots A_n \to B$$

that is coded using only DT-functions and functions that have a cps-counterpart (including `if`) has a cps-counterpart

$$\texttt{f_cps} : A'_1 \to \ldots \to A'_n \to (B' \to \texttt{formula}) \to \texttt{formula}$$

such that the following evaluation/interpretation diagram commutes:



This means we can successively provide a cps-counterpart to any function used to program `proj_acf` and `prof_rcf`, including non DT-functions. Since the correctness lemma of the direct counterpart of a DT-function is much shorter and easier to use than the one of its cps-counterpart, we use cps-counterparts only for non DT-functions and we keep using direct counterparts for DT-functions. A consequence of this is that `proj_acf` and `prof_rcf` have cps-counterparts.

Example

Let us study the example of the `test` function that tests whether the leading coefficient of a polynomial is greater that 0:

```
Definition test (p : {poly R}) : bool := 0 < lead_coef p.
```

We now build the formal counterpart `testF` of test. It suffices to call `LeadCoef` on `p` and give as a continuation the function that tests if a term is greater than 0.

```
Definition Test (p : polyF) : formula :=
                          LeadCoef p (fun x => Lt (Const 0) x).
```

Let us remark that although `test` uses non DT-functions in its code, since its return type is `bool`, it remains a DT-function. This is a general fact: any function returning a Boolean is a DT-function which direct counterpart returns a formula.

Now, the following meta-theorem suffices to conclude that `proj_rcf` and `proj_acf` are DT-functions.

Theorem 12.2. *If a function g with return value in `bool` has a cps-counterpart, then it is a DT-function, which means it also has a direct counterpart.*

Proof. If g has type

$$A_1 \to \dots A_n \to \texttt{bool},$$

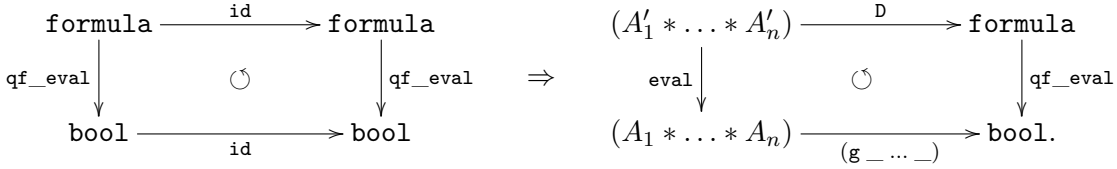
its cps-counterpart `G` has type

$$A'_1 \to \dots \to A'_n \to (\texttt{formula} \to \texttt{formula}) \to \texttt{formula}.$$

Now, we pose `(D` $a_1 \dots a_n$ `:= G` $a_1 \dots a_n$ `id)`. The function `D` has type

$$A'_1 \to \dots \to A'_n \to \texttt{formula}$$

and we can prove it is the direct counterpart. Indeed it suffices to replace `k` and $\bar{k}$ by `id` in the diagram of commutation of the cps-counterpart, and the left hand side becomes true.



□

Part IV

Conclusion and perspectives

Conclusion and perspectives

On Coq and SSReflect

It has been a real pleasure to work using Coq and SSReflect. Indeed, the latter was a wonderful toolbox, able to help me solve numerous problems, not only by providing the appropriate database of lemmas, but also by suggesting a methodology, and providing a concise and powerful tactic language and a great *search* functionality.

Formalizing mathematics taught me a lot of mathematics I thought I understood, but which I rediscovered under a new angle, the one where you cannot possibly miss any detail. It helped me rethink the notion of what I believed was a proof and it definitely taught me never to trust anymore the capacity of humans to check the validity of proof: I now only trust them for their intuition. However, I also learned that you should think very hard before trying to prove something, because with Coq, there is no "almost truth": if your statement is not provable, you can lose a lot of time before noticing it. Moreover, if a lemma is not stated in the way you will use it, then you will also lose a lot of time restating it correctly and doing the proof again. Despite this, Coq has also been a great tool to toy with and discover what was the right statement.

Notations and implicit data

There is a balance to keep between conciseness, expressiveness and readability. While the user might gladly forget that he is handling algebraic structures and may only want to see the carrier and some of the algebraic operations when he needs to, the system must keep track of all the information. This kind of situation makes mandatory to hide some information from the system to the user, which is what implicit arguments and notations are for. However, hiding too much information can also affects the readability by a human. The constraints on readability are different from the ones that occur for programming (without proofs) and depends also on the context. For programming, overloading can very convenient, but for proving it can be essential to be able to tell apart two terms that look the same but are not.

Proof engineering

During the three last years, I have been formalizing various results in Coq, either to toy with them or to achieve a greater result. I pushed those results to various stages of development, which I would classify into three categories.

1. The first stage is the one of unfinished developments, that I aborted because I had seen enough of, or because I did not believe it could work out in the end, or simply because I did not have enough time.

2. The stage of those which could be considered as finished, because I reached the result I wanted to prove, but that are not reusable by others, not even me sometimes, not as such. This could happen either because there is no documentation, or just because it probably does not contain yet what is necessary to prove a different result than the one it was designed for.

3. Accordingly to the latter drawback, the final stage is to make the development usable by other users, and this is in my opinion both the most difficult, essential and interesting work. It not only consists in providing enough lemmas, but also in finding the most appropriate form for their statement, finding a better abstraction, and often writing better and shorter proofs. However, I learned that reaching that stage does not mean that the development is over.

Unsurprisingly, reaching each next step for each development always gave me a lot of satisfaction, but the last one was the most thrilling.

The parts of development I pushed to the last stage so far are the numeric hierarchy (Chapter 4) and the tool for quotienting (Chapter 6). Quotients are only used once in my thesis work but they are used by other persons and in some of my own unfinished developments. As for the numeric hierarchy, it was necessary to reach the final stage because it is used in (almost) everything we formalized, but also in the proof of the Feit-Thompson Theorem. It took us a year to reach a stable version: choosing the right interfaces, their organization, the amount of theory to provide, and the naming convention. Also, since integers and rationals are implementations of interfaces from the numeric hierarchy they can also be considered as in final state. Note that the development on polynomial division (end of Section 3.1) also reached this final stage, but not from my hand. Because they reached this stage, all of the developments cited in this paragraph got included in the latest release of SSReflect (v1.4).

Note that the contributions of creating real and algebraic numbers and programming a quantifier elimination procedure can also be seen as the implementation of given interfaces. Indeed, real and algebraic numbers respectively implement discrete real and algebraically closed field interfaces, and the quantifier elimination procedure can be seen as an achievement to implement an interface for fields with decidable first order theory. In this sense, they could also be considered as finished. However, the material we used to write those implementations is often not specific to them, and other formalizations could benefit from their further development.

To conclude with this classification, nothing we presented here is at the unfinished stage. Indeed, the rest of this thesis work can be classified in the second category, which leads us to describe developments that could extend or use the work from this thesis.

Possible applications and extensions

We noticed that by extending the big enough tactic to handle intersections of predicates, we could handle "small enough" neighborhoods. This could simplify lots of proofs in the development about polynomial analysis in discrete real closed fields (Chapter 9). We also think it could help to formalize a theory about limits of functions and the study of equivalence, of "little o" and "big O" notations and of limited developments. Indeed, the theory of the asymptotic behaviour of functions relies on reasoning of the form that "big/small enough" handles. We also hope it may also be used to implement non standard analysis, or at least provide a tool to reason like in non standard analysis but in a certified framework, not prone to human error. A development of asymptotic reasoning (using delimited developments or non standard analysis) would also simplify further some definitions and proofs in the polynomial analysis on real closed fields, like for example the notions of *jump* or of *sign on the right of a polynomial*, which could be formalized more directly and would have properties deriving directly from the theory of asymptotic reasoning.

As another extension, we may want to formalize one of the proofs of the fundamental theorem of algebra from Section 8.1. Even if we already used a constructive proof for this theorem, it could be useful to prove it using symmetric polynomials or the universal decomposition algebra, which are two interesting object for other formalizations of mathematics and in computer algebra.

Currently, the quantifier elimination procedure is written twice: on mathematical objects and on reified terms. Moreover the proof that the former objects are the interpretation of the latter involve writing pieces of code on mathematical objects each time a recursive function is involved. This leads to a large amount of duplicated code which Coq is unable to infer automatically for now, so that the user (me) had to enter it by hand. A solution for this would be to reify the quantifier elimination program itself and from it generate both the Coq function operating on mathematical objects and the Coq function operating on reified terms. This would considerably shorten the size of the code and ease its maintenance, since there would be only one code to maintain.

The proof of quantifier elimination we made is semantic: we prove that the output formula – which is quantifier-free – has the same interpretation as the input formula. However, we did not prove that the equivalence is provable in a natural deduction or sequent calculus system (for example). Since we did not do a deep formalization of model theory, we consider that models of real closed fields are instances of the interface of real closed field from the numeric hierarchy (described in Chapter II). But this interface is the one of *discrete* real closed fields, which means equality and comparison are decidable. This makes sense, since without the decidability of the atoms of the theory (here equality and comparison), quantifier elimination does not imply the decidability of the first-order theory. What is more debatable is the fact that the equality cannot be a setoid equality in our framework, which excludes the Cauchy reals as a discrete real closed field. At least we know that real algebraics form an instance of the discrete real closed field interface, which means that this quantifier elimination proof is not groundless: we can eliminate quantifiers from a first-order formula with variables that are real algebraic numbers.

However, we hope that a formalization of Gödel completeness theorem (if it is possible to prove it constructively) would provide a syntactic quantifier elimination

proof. Indeed, we proved semantic quantifier elimination is valid in any discrete model of real closed fields. This is classically equivalent to being valid in any model of real closed fields. In particular, we hope that it is possible to do the construction of a syntactic model (à la Herbrand or Henkin) using the existence of real algebraic number, and to derive the provability of the equivalence between a first-order formula and its quantifier free counterpart. From such a proof it may be easy to derive semantic quantifier elimination on any model, discrete or not.

On automation

In our work we did not use any decision procedure, although we studied objects for which decision procedures exist: the theory of rings expressions [17, 48] and of linear arithmetic [10]. One of the first reason is purely technical: rings and natural number expressions in SSReflect are not expressed in a way that the implementation of the decision procedure could handle. However, the main reason — which is also a reason why we did not take much time to try to fix the previous problem — is that we did not need decision procedures to complete our proofs. Most of the problems we encountered that belonged to these decidable theories were often so simple a single line of tactics could solve them. As already mentioned in Section 4.3 and Section 11, most of the problems that seemed to require the use of one of those two decision procedures could be solved by adding a simple abstraction layer (such as intervals for Section 4.3, and variations for Section 11). If those abstraction layers were only designed to avoid the use of decision procedures, it would not have been a very good reason for their introduction. However, those abstraction also made the code more readable, and the proofs smaller and more robust (even those which did not need decision procedures at all).

Nevertheless, decisions procedures can be useful and we think a natural improvement of the current state of the SSReflect library, and more particularly the algebraic and numeric hierarchies, is to provide a connection with Coq decision procedures. The current development on numeric domains — which does not rely on decision procedures — could even be used to bootstrap the certification of another linear arithmetic decision procedure, if one wants to re-implement one.

On effective computation

In this thesis we show how we constructed the real algebraic closure and the complex algebraic closure of any discrete Archimedean real closed field. There were many different choices for this formalization, we made the choices we believed would be faster to implement, given the current state of the art in formalization of mathematics. The datatypes we used are naive and the arithmetic operations and decidable predicates and relations we implemented are not efficient. For example, integers have been defined as the disjoint sum of positive and negative unary natural numbers. Because of this, computation on integers and rationals only work for small examples. Moreover, in the SSReflect library, some operations are locked, which means that a piece of code has been made opaque in order to stop Coq from computing it. One of the reason why we want Coq not to compute is because proofs are often made at some level of abstraction, and letting Coq reduce a term could break

the abstraction. For example, when using the ring addition on integers, if we let the system reduce the ring addition to the specific addition of integer, we lose the information it is a *ring* operator and thus we lose the possibility to use ring theory to solve our specific problem. Another reason for locking is that some problems make the current kernel of Coq diverge. We do not know precisely whether the reason for locking is fundamental or only necessary because of Coq current implementation.

Anyway, in our experience, naive structures and naive algorithms are easier to study than their efficient counterparts. Because of this, it is not necessarily a problem for naive structures to be locked, but it calls for a systematic methodology for translating naive structures and operations into their efficient counterparts, run them and get back to naive structures. This looks very much like a form of large scale reflection. This approach has been investigated recently [35], and seems very promising.

We hope that a generalization of this approach may help forging and certifying a computable structure for real algebraic numbers. The latter approach gives a generic solution for switching to a "computation mode" in Coq, yet the implementation of the computable structures and efficient algorithms remain specific to the problem, here algebraic numbers. For real algebraic numbers we think that one may use a fast implementation of real numbers which could be based on

- dyadic numbers [59] if possible, in order to compute approximations;
- a better theory of interval arithmetic, to get more precise bounds for polynomials, with possible inspiration from [69];
- potentially Newton's method instead of dichotomy for approximating roots. Newton's method has already been formalized and proved correct in Coq [68, 58];
- Newton sums to compute the specific resultant that occur in the computation of arithmetic operations on algebraic Cauchy reals, as described in graduate courses [16].

The quantifier elimination procedure we presented can be seen as a naive version of Collins' Cylindrical Algebraic Decomposition (cf [28]), except Cylindrical Algebraic decomposition algorithms do not only output finite sets of cells, but also (multidimensional) algebraic points from each cell. By evaluating the polynomials in each point we can reconstruct the quantifier-free formula if we want to. However, they share the same ideas of computing the Cauchy index in an algebraic way, and the linear transformation of Tarski queries with different algorithms relying on more advanced mathematical object. Anyway, they share a strong common ground on the theory of polynomials on real closed fields. The computation of the Cauchy index uses subresultants instead of pseudo remainder sequences. And the computation of the number of root satisfying some constraint uses an adapted matrix of smaller dimension in the number of polynomials instead of a 3^n-square matrix, as detailed in our reference on the subject [5]. Moreover, expressing the algebraic points in each cell could might be done using algebraic numbers: using the current naive implementation for the theory and the efficient one in the Cylindrical Algebraic Decomposition procedure.

Applications using effective computation

In the context of the proof of the Feit-Thompson Theorem by the mathematical components team [74], Russell O'Connor formalized Galois theory constructively in SSReflect.[1] We hope that efficient algebraic numbers could lead to the effective computation of Galois groups of polynomials.

The quantifier elimination procedure as described in Section 12.4 is programmed in continuation passing style. In each function, the continuation is executed on the sub-cells that are created. Each sub-cell is stored as an incomplete formula in the call stack of the function. Instead of letting the stack store this result, we could store it manually using a well-defined monad. This *cell*-monad would hold a finite set of cells and accumulate constraints to refine this set. It could even be optimized to avoid the duplication of identical constraints.

[1] This is not published.

List of Figures

1.1 Large scale reflection . . . 12
1.2 The ring tactic . . . 12
1.3 Small scale reflection . . . 13
1.4 Boolean reflection . . . 13

2.1 Packaging of mathematical structures . . . 22
2.2 The algebraic hierarchy . . . 23
2.3 Signatures of SSReflect algebraic structures . . . 24

4.1 Extension of the hierarchy with Numeric and Real interfaces . . . 32
4.2 Axiom of the Numeric mixin . . . 33
4.3 Axioms to produce directly a Real structure . . . 33
4.4 Naive case analysis on an order statement . . . 37
4.5 A three constructor inductive instead of `sumbool` . . . 38
4.6 Three way case analysis with a simple inductive . . . 38
4.7 Definition of `ltrgtP` . . . 39
4.8 Three way case analysis using an inductive family . . . 39
4.9 A non structured interval membership goal . . . 40
4.10 An interval membership goal. . . . 41
4.11 Generating rewrite rules for intervals . . . 42
4.12 Subinterval decision procedure . . . 43

5.1 The construction of algebraic Cauchy reals . . . 47

6.1 Quotients without equivalence relation . . . 63
6.2 Quotient interface . . . 63

7.1 Encoding algebraic numbers as a choice structure . . . 76
7.2 The construction of real algebraic numbers . . . 78
7.3 Alg is the real algebraic closure . . . 79

8.1 Equivalent definitions for real closed fields . . . 82

11.1 Cauchy index on a bounded interval 119

12.1 Pseudo remainder tree 126
12.2 Differences of sign changes on C_7 127

Bibliography

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, pages 64–69, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in Unification. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 84–98, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 2006.

[4] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003. Special Issue on Logical Frameworks and Metalanguages.

[5] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*, volume 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

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

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[8] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *Theorem Proving in Higher-Order Logics*, volume 5170 of *LNCS*, pages 86–101, 2008.

[9] Yves Bertot, Fréderique Guilhot, and Assia Mahboubi. A formal study of Bernstein and coefficients polynomials. *Mathematical Structures in Computer Sciences*, 2011.

[10] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs (TYPES 06), Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, 2006.

[11] Sidi Ould Biha. Formalisation des mathématiques : une preuve du théorème de Cayley-Hamilton. In *Journées Francophones des Langages Applicatifs*, January 2008.

[12] Sidi Ould Biha. Finite groups representation theory with Coq. In *Mathematical Knowledge Management*, 2009.

[13] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.

[14] Jack Bochnak, Michel Coste, and Marie-Françoise Roy. *Real Algebraic Geometry*, volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1998.

[15] Alin Bostan. *Algorithmique efficace pour des opérations de base en Calcul formel*. PhD thesis, École polytechnique, 2003.

[16] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Bruno Salvy, and Éric Schost. Algorithmes efficaces en calcul formel, january 2012. En cours de rédaction, Notes de cours 2-22 du MPRI, année 2010-2011 http://algo.inria.fr/chyzak/mpri/poly-20120112.pdf.

[17] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0014565.

[18] Auguste Cauchy. Calcul des indices des fonctions. *Journal de l'École polytechnique*, 15(25):176–229, 1832.

[19] Claude Chevalley and Henri Cartan. Schémas normaux; morphismes; ensembles constructibles. In *Séminaire Henri Cartan*, volume 8, pages 1–10. Numdam, 1955-1956. http://www.numdam.org/item?id=SHC_1955-1956__8__A7_0.

[20] Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, number 2646 in LNCS, pages 95–107. Springer, 2003.

[21] Cyril Cohen. Types quotients en Coq. In Hermann, editor, *Actes des 21ème journées francophones des langages applicatifs (JFLA 2010)*, Vieux-Port La Ciotat, France, January 2010. INRIA.

[22] Cyril Cohen. Construction des nombres algébriques en Coq. In *Proceedings of JFLA2012*, 2012. http://perso.crans.org/cohen/work/realalg/ (to appear).

[23] Cyril Cohen. Construction of real algebraic numbers in Coq. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, Princeton, États-Unis, August 2012. Springer.

[24] Cyril Cohen and Thierry Coquand. A constructive version of Laplace's proof on the existence of complex roots. http://hal.inria.fr/inria-00592284/PDF/laplace.pdf.

[25] Cyril Cohen and Assia Mahboubi. A formal quantifier elimination for algebraically closed fields. In *Symposium on the Integration of Symbolic Computation and Mechanised Reasoning, Calculemus Intelligent Computer Mathematics*, volume 6167 of *Computer Science*, pages 189–203, Paris France, 06 2010. Springer. The final publication is available at www.springerlink.com.

[26] Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1:02):1–40, February 2012.

[27] Paul J. Cohen. Decision procedures for real and p-adic fields. *Communications on Pure and Applied Mathematics*, 22(2):131–151, 1969.

[28] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition–preliminary report. *SIGSAM Bull.*, 8:80–90, August 1974.

[29] Thierry Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[30] René Cori and Daniel Lascar. *Logique mathématique 1 - Calcul propositionnel ; algèbre de Boole ; calcul des prédicats*. Dunod, Paris, 2003. réédition (avec corrections) de l'édition Masson 1993.

[31] Pierre Courtieu. Normalized types. In *Proceedings of CSL2001*, volume 2142 of *Lecture Notes in Computer Science*, 2001.

[32] Luis Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 108–126. Springer–Verlag, 2003.

[33] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 85–95, Reunion Island (France), November 2000. Springer.

[34] Harm Derksen. The Fundamental Theorem of Algebra and Linear Algebra. *American Mathematical Monthly*, 110(7):620–623, August/September 2003.

[35] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in Coq. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer Berlin Heidelberg, 2012.

[36] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2009 proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[37] François Garillot. *Generic Proof Tools And Finite Group Theory*. Phd thesis, École polytechnique X, December 2011.

[38] Carl Friedrich Gauss. Another new proof of the theorem that every integral rational algebraic funcion of one variable can be resolved into real factors of the first or second degree. 1815, translated by P. Taylor and B. Leak (1983).

[39] Herman Geuvers and Milad Niqui. Constructive Reals in Coq: Axioms and Categoricity. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '00, pages 79–95, London, UK, 2002. Springer-Verlag.

[40] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2000.

[41] Georges Gonthier. The four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[42] Georges Gonthier. Advances in the Formalization of the Odd Order Theorem. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *ITP*, volume 6898 of *Lecture Notes in Computer Science*, page 2. Springer, 2011.

[43] Georges Gonthier. Point-free, set-free concrete linear algebra. In *Interactive Theorem Proving, ITP 2011 Proceedings*, Lecture Notes in Computer Sciences. Springer, 2011.

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

[45] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA, 2007.

[46] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A small scale reflection extension for the Coq system*. INRIA Technical report, http://hal.inria.fr/inria-00258384.

[47] Georges Gonthier and Enrico Tassi. A Language of Patterns for Subterm Selection. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 361–376. Springer Berlin Heidelberg, 2012.

[48] Benjamin Gregoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS 3603*, pages 98–113. Springer, 2005.

[49] D. Yu Grigor'ev. Complexity of deciding tarski algebra. *Journal of Symbolic Computation*, 5(1-2):65–108, February 1988.

[50] D. Yu. Grigor'ev and N. N. Vorobjov, Jr. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5(1-2):37–64, February 1988.

[51] John Harrison. Formalizing an analytic proof of the Prime Number Theorem (dedicated to Mike Gordon on the occasion of his 60th birthday). *Journal of Automated Reasoning*, 43:243–261, 2009.

[52] Michael Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, pages 4–8, 1998.

[53] Joos Heintz, Marie-Françoise Roy, and Pablo Solernó. Sur la complexité du principe de tarski-seidenberg. *Bulletin de la Société Mathématique de France*, 118(1):101–126, 1990.

[54] Wilfried Hodges. *A shorter model theory*. Cambridge University Press, 1997.

[55] Martin Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, University of Edinburgh, 1995.

[56] Lars Hörmander. *The analysis of linear partial differential operators*, volume 2. Springer-Verlag, Berlin etc., 1983.

[57] Paul Hudak, John Peterson, and Joseph Fasel. Gentle introduction to haskell, version 98, June 2000. http://www.haskell.org/tutorial/.

[58] Nicolas Julien and Ioana Pasca. Formal Verification of Exact Computations Using Newton's Method. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 408–423, 2009.

[59] Robbert Krebbers and Bas Spitters. Computer certified efficient exact reals in Coq. In *Conference on Intelligent Computer Mathematics, CICM 2011 Proceedings*, Lecture Notes in Artifical Intelligence. Springer, 2011.

[60] Serge Lang. *Algebra*. Graduate texts in mathematics. Springer, 2002.

[61] Pierre-Simon Laplace. Leçons de mathématiques données à l'École normale en 1795. In *Oeuvres complètes de Lapalace. Tome XIV*, pages 10–177. Gauthier-Villars (Paris), 1878–1912.

[62] Henri Lombardi and Claude Quitté. *Algèbre commutative, Méthodes constructives*. Calvage et Mounet, 2011.

[63] Ray Mines, Fred Richman, and Wim Ruitenburg. *A course in constructive algebra*. Universitext (1979). Springer-Verlag, 1988.

[64] Tobias Nipkow, Clemens Ballarin, and Jeremy Avigad. Isabelle/HOL: Theory SetInterval. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/SetInterval.html.

[65] Russel O'Connor. *Incompleteness & Completeness, Formalizing Logic and Analysis in Type Theory*. PhD thesis, Radboud University Nijmegen, Netherlands, 2009.

[66] Russell O'Connor. Certified exact transcendental real number computation in Coq. In Otmane Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 246–261. Springer Berlin / Heidelberg, 2008.

[67] Sidi Ould Biha. *Composants mathématiques pour la théorie des groupes*. PhD thesis, Ecole doctorale STIC, Université de Nice Sophia-Antipolis, February 2010.

[68] Ioana Pasca. Formal Proofs for Theoretical Properties of Newton's Method, 2010. INRIA Research Report RR-7228.

[69] Ioana Pasca. Formally verified conditions for regularity of interval matrices. In *17th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning, Calculemus 2010*, volume 6167 of *Lecture Notes in Artificial Intelligence*, pages 219 – 233. Springer, 2010.

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

[71] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[72] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programmi ng Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. technical report CMU-CS-89-209.

[73] Loïc Pottier. Quotients dans le CCI. Report RR-4053, INRIA, November 2000.

[74] The Mathematical Components Project. SSReflect extension and libraries. http://www.msr-inria.inria.fr/Projects/math-components/index_html.

[75] James Renegar. On the computational complexity and geometry of the first-order theory of the reals. part i-iii. 13(3):255 – 352, 1992.

[76] Julia Robinson. Definability and decision problems in arithmetic. *Journal of Symbolic Logic*, 14:98–114, 1949.

[77] Amokrane Saibi. Typing algorithm in type theory with inheritance. In *Principles of Programming Languages, POPL 1997 proceedings*, pages 292–301, 1997.

[78] A. Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 60(2):pp. 365–374, 1954.

[79] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009.

[80] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2008 proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[81] Bas Spitters and Eelis van der Weegen. Type Classes for Mathematics in Type Theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.

[82] Pierre-Yves Strub. Coq Modulo Theory. In Anuj Dawar and Helmut Veith, editors, *19th EACSL Annual Conference on Computer Science Logic Computer Science Logic, CSL 2010, 19th Annual Conference of the EACSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 529–543, Brno Czech Republic, 2010. Springer.

[83] Alfred Tarski. A Decision Method for Elementary Algebra and Geometry. Manuscript. Santa Monica, CA: RAND Corp., 1948. Republished as A Decision Method for Elementary Algebra and Geometry, 2nd ed. Berkeley, CA: University of California Press, 1951.

[84] The Coq Development Team. *The Coq Proof Assistant, Reference Manual*. http://coq.inria.fr.

[85] Laurent Théry, Pierre Letouzey, and Georges Gonthier. Coq. In Wiedijk [88], pages 28–35.

[86] Thomas Braibant and Damien Pous. Deciding Kleene Algebras in Coq. *Logical Methods in Computer Science*, 8(1), 2012.

[87] Eelis van der Weegen, Bas Spitters, and Robbert Krebbers. Math classes. http://math-classes.org.

[88] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

Index

Axiom, 17
P.[x], 26
ProjTermAcf, 108
ProjTermRcf, 109
archiFieldType, 32
bool, 11
bounding_poly, 114
cauchy_bound, 94
cauchy_boundP, 94
cauchymodP, 48
changes, 121
changes_horner, 121
closedFieldType, 104
constraints, 116
constraints1, 115
count_acf, 113
count_rcf, 123
diffP, 51
eqType, 15
eq_op, 15
eqn, 15
eqnP, 16
false, 11
(_ \in _), 16
is_true, 13
itv_splitU, 43
jump, 120
lboundP, 50
le_modP, 51
leq_xor_gtn, 16
mods, 121
mvt, 94
nat, 13, 15
nat_of_bool, 120
predType, 16
projT1, 15
proj_acf, 111, 114
proj_rcf, 111, 124
proj_real, 107, 109
rcfType, 32
rdivp, 26
reflect, 14
rgcdp, 27
rgdcop, 113
rmodp, 26
rolle, 93
rolle_weak, 94
root, 26
roots, 96
rscalp, 26
seq, 16
sizeY, 27
{subset A <= B}, 16
swapXY, 27
taq, 116
true, 11, 13
uboundP, 49
xchoose, 18, 50, 51, 56

Axiom
    Coq, 17
    Mathematical, 17

Base Type, 72

Big Enough, 54
Bounding Polynomial, 114

Canonical, 20
    Structure, 20
    Surjection, 62
Canonically, 75
Carrier, 18
Cauchy
    Bounds, 120
    Index, 119
Cauchy Bound, 94
Cell, 126
Class, 18, 62
Coding Property, 71, 77
Coercion, 20
Commutative Group, 19

Decidable
    Equality, 15
Discrete, 18
Domain
    Integral, 19
    Numeric, 31
    Real, 32

Existential
    Variable, 55

Field, 19
    Decidable, 19
    Algebraically Closed, 19, 102
    Archimedean, 32
    Formally Real, 34
    Numeric, 31
    Real, 32
    Real Closed, 32, 103
Field, record, 18
Fundamental Theorem of
    Algebra, 81
    Symmetric Polynomials, 84, 85

Greatest Common Divisior (Pseudo), 27
Greatest divisor coprime to, 112

Hierarchy
    Algebraic, 31
    Numeric, 31
Horner's Algorithm, 26

Inductive
    Family, 16, 36
Integers, 20
Interface, 18

Jump, 119

Lifting
    Quotient, 64
Ltac, 55

Mean Value Theorem, 94
Mixin, 18
Module, 19

Projection, 18
Proof Irrelevance, 15, 76
Prop, 13
Pseudo-remainder, 27
    Sequence, 120

Quotient
    Structure, 62
    Type, 62

Record, 18
    Type, 18
Reflection, 11
    Boolean, 13
    Large Scale, 12
    Small Scale, 12, 13
Representative, 62
Resultant, 27
Ring, 19
Rolle's Theorem, 93
Root
    Predicate, 26
    Sequence, 96

Setoid, 49, 61
Signature, 18
Structure, 18
Subtype, 15

Tarski Query, 116
Tarski query, 115
Type Class, 21

Unification, 20