Logical Methods in Computer Science
Vol. 8 (1:02) 2012, pp. 1–40
www.lmcs-online.org

Submitted May 16, 2011
Published Feb. 16, 2012

FORMAL PROOFS IN REAL ALGEBRAIC GEOMETRY: FROM ORDERED FIELDS TO QUANTIFIER ELIMINATION

CYRIL COHEN [a] AND ASSIA MAHBOUBI [b]

[a,b] INRIA Saclay – Île-de-France, LIX École Polytechnique, INRIA Microsoft Research Joint Centre,
e-mail address: cohen@crans.org, Assia.Mahboubi@inria.fr

Abstract. This paper describes a formalization of discrete real closed fields in the Coq proof assistant. This abstract structure captures for instance the theory of real algebraic numbers, a decidable subset of real numbers with good algorithmic properties. The theory of real algebraic numbers and more generally of semi-algebraic varieties is at the core of a number of effective methods in real analysis, including decision procedures for non linear arithmetic or optimization methods for real valued functions. After defining an abstract structure of discrete real closed field and the elementary theory of real roots of polynomials, we describe the formalization of an algebraic proof of quantifier elimination based on pseudo-remainder sequences following the standard computer algebra literature on the topic. This formalization covers a large part of the theory which underlies the efficient algorithms implemented in practice in computer algebra. The success of this work paves the way for formal certification of these efficient methods.

1. Introduction

Most interactive theorem provers benefit from libraries devoted to the properties of real numbers and of real valued analysis. Depending on the motivation of their developers these libraries adopt different choices for the definition of real numbers and for the material covered by the libraries: some systems favor axiomatic and classical real analysis, some other more effective versions.

The present paper describes the formalization in the Coq system [36, 4] of basic real algebraic geometry, which is the theory of sets of roots of multivariate polynomials in a real closed field, as described for instance in [2]. One of our main motivations is to provide formal libraries for the certification of algorithms for non-linear arithmetic and for optimization problems. The theories we formalize apply to any instance of real closed field, defined as an ordered field in which the intermediate value property holds for polynomial functions. Up to our knowledge, it is the first formal library on real numbers developed at

1998 ACM Subject Classification: F.4.1.

Key words and phrases: Formal proofs, Coq, quantifier elimination, small scale reflection, real algebraic geometry, real closed fields.

[a,b] This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

DOI:10.2168/LMCS-8 (1:02) 2012

this level of abstraction. Such an interface of real closed field can be instantiated by a classical axiomatization of real numbers but also by an effective formalization of real algebraic numbers.

We start with a formalization for the elementary theory of polynomial functions, obtained as a consequence of the intermediate value property. This part of our work is largely subsumed by the libraries available for real analysis, which study continuous functions in general and not only polynomials, but is imposed by our choice to base this work on an abstract structure.

Then we formalized a proof of quantifier elimination for the first order theory of real closed fields. Since the original work of Tarski [35] who first established this decidability result, many versions of a quantifier elimination algorithm have been described in the literature. The first, and elementary, algebraic proof might be the one described by Hörmander following an idea of Paul Cohen [21, 7]. The best known algorithm is Collin's Cylindrical Algebraic Decomposition algorithm [13], whose certification is our longer-term goal. All these algebraic proofs rely on the same central idea : 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 study of the (multidimensional) roots of the polynomials in $\mathcal{Q}$ should provide all the information necessary to validate or invalidate any first order property which can be expressed by a closed first order statement whose atoms are constraints on the polynomials in $\mathcal{P}$. The recursive application 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 used repeated Euclidean divisions and simple derivatives. The breakthrough introduced by the Collins' algorithm is due to a clever use of better remainder sequences, namely subresultant polynomials, and of partial derivatives in order to improve the complexity of the projection.

In this paper, we describe a quantifier elimination algorithm with a naive complexity (a tower of exponentials in the number of quantifiers). We follow the presentation given in the second chapter of [2], based on the original method of Tarski [35]. This algorithm is more intricate than the one of Hörmander, and closer to Collins' one with respect to the objects it involves, hence our choice. In particular, we hope that the material we describe here will significantly ease the formalization of the correctness proof of Collins' algorithm. 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.

To the best of our knowledge, the present paper describes several original contributions: first, we describe an interface of real closed field with decidable comparison which is integrated in an existing larger algebraic hierarchy. This interface comes with a library describing the elementary theory of real closed fields. Then we formalize real root counting methods based on pseudo-remainder sequences. Finally we formalize a complete proof of quantifier elimination for the theory of real closed fields with decidable comparison. From this proof we deduce the decidability of the full first order theory of any instance of real closed field with decidable comparison.

Since the formalization is modular and based on an abstract interface, all these results become immediately available for any concrete instance of the interface, like an implementation of real algebraic numbers or a classical axiomatization of real numbers. All these proofs are axiom-free and machine-checked by the Coq system.

The paper is organized as follows: in section 2 we summarize some aspects of existing libraries we are basing our work on, including in particular an existing hierarchy of algebraic structures. We then show in section 3 how we extend this hierarchy with an interface for real closed fields. In particular this includes an infrastructure for real intervals. Section 4 is devoted to the elementary consequences of the intermediate value property, and culminate with the formalization of neighborhoods of roots of a polynomial. In section 5, we describe the core of the algorithm. We introduce pseudo-remainder sequences, Cauchy indexes and Tarski queries, and show how to combine these objects in an effective projection theorem. In section 6 we briefly describe a deep embedding of the first order formulas on the signature of ordered fields and the implementation of the formula transformation eliminating quantifiers. Formalizations described in this section are based on a previous work [12] which we explain here in more details and adapt from the case of algebraically closed fields to the case of real closed fields. We conclude by describing some related work and further extensions.

2. SSReflect libraries

This section is devoted to a brief overview of the main features of the SSReflect libraries we will be relying on in the present work. The material exposed in this section comes from the collective project [30] of formalization of the Odd Order Theorem [3, 29]. Both authors of the present article are active in the latter project. Yet the content described in this section has been developed by many people from the Mathematical Component project and is not specific to the formalization we describe in the present article.

2.1. On small scale reflection and its consequences.

SSReflect libraries rely extensively on the small scale reflection methodology. In the Coq system, proofs by reflection take benefit from the status of computation in the Calculus of Inductive Constructions to replace some deductive steps by computation steps. This has been quite extensively used to implement proof-producing decision procedures, following the pioneering work of [8]. The small scale variant of this method addresses a different issue: its purpose is not to provide tools to solve goals beyond the reach of a proof by hand by the user. Instead, small scale reflection favors small and pervasive computational steps in formal proofs, which are interleaved with the usual deductive steps. The essence of this methodology lies in the choice of the data structures adopted to model the objects involved in the formalization. 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: `O` for the zero constant and `S` for the successor. 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 compared.

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

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

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 that `(leq 2 2)` and a proof of `(leq 2 4)` consists 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. The motivation for small scale reflection is however 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.

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. As suggested by this example, the small scale reflection methodology models a fragment of propositions as booleans, as opposed to logical statements in the `Prop` sort of Coq. This fragment corresponds to the propositions on which excluded middle holds: one says that the `bool` datatype reflects this fragment and we call this formalization choice "boolean reflection". Any boolean value `b` can be interpreted in `Prop` by the statement `(b = true)`. This remark is implemented by declaring the coercion:

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

which is automatically and silently inserted by the Coq coercion mechanism when needed: this can be considered as a simple explicit subtyping mechanism. From now on, we will implicitly use booleans as propositions in code excerpts like we would do in the standard input/display mode of the Coq system once the previous coercion has been declared.

Two boolean expressions represent equivalent statements if their truth tables are the same. In the same way, two expressions returning a boolean represent equivalent statements if they have the same value when instantiated with the same parameters.

For instance we prove the theorem:

```
Lemma leqNgt : forall m n : nat, (m <= n) = ~~ (n < m).
```

where `~~` is the boolean negation, where the notation `(n <= m)` stands for `(leq n m)` and the notation `(n < m)` for the strict comparison on natural numbers. This property would have been modelled by a logical equivalence if we were working with the standard Coq library `le` predicate. Adopting boolean reflection even increases the importance of rewriting steps in a proof: local transformations of a boolean goal are performed by rewriting lemmas like `leqNgt` (see examples in section 3.3).

Due to this extensive use of rewriting rules, the SSReflect tactic language provides an advanced `rewrite` tactic to chain rewriting, select occurrences using patterns, and get rid of trivial side conditions (see [18] for more details).

The SSReflect library also provides support for the theory of container datatypes, 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 comparison (see section 2.2), 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 [17].

2.2. **Interfaces.** In this section we call a (mathematical) *structure* a carrier closed under some operations satisfying a given list of specifications. The list of the constants and operations used in the characterization of the structure is the signature of the structure. For instance, the signature of the structure of field consists of two constants 0 and 1 and three operations: the additive and multiplicative binary laws and the unary additive inverse. The subtraction operation, though always definable in any ring, is not part of the signature but is defined using the appropriate combination of addition and opposite. We use the term of *interface* for the implementation of the definition of a mathematical structure in the Coq proof assistant. Such an interface can be a first class object like a record type: in this case, the instances of the structure are the inhabitants of this type. Interfaces can also be implemented by a second class object like a module type. In the latter case, the interface is not a defined object and cannot be used as quantification carrier.

The revised published proof [3, 29] of the Odd Order Theorem is not only about finite groups: it convenes a wide variety of mathematical structures, together with their signatures, usual syntactic notations, and elementary theories. The algebraic hierarchy organizing the interplay between this collection of operators and properties is therefore a critical component of the SSReflect libraries. The aim of such a hierarchy is to support the automated inference of mathematical properties and the safe overloading of notations.

2.2.1. *Purpose.* Automating the inference of mathematical properties is of a crucial importance when working with large scale mathematical libraries. There is no really clever trick to help a user proving that integers are equipped with a ring structure, or that the product of two arbitrary ring structures can always be equipped with a ring structures. But if the user has already provided this effort, it is not reasonable to require from the same user some manual input or extra formalization in order to apply ring theory to pairs of integers. The system is hence expected to *infer* that a pair, or any combination of canonical constructions of rings, applied to any known rings, leads to a ring structure. An important class of such canonical constructions are carried by structure morphisms: the image and preimage of a group by a group morphism is itself a group, etc. Again, this is captured by the design of appropriate interfaces for structure morphisms. As a consequence, in the hierarchy we describe, these structures are modelled by first class objects like record types, as opposed to the mechanism offered by the module system of Coq[4], since the specification of morphisms involve quantifications over the instances of the related structure. Last, an important issue which should be addressed by such a hierarchy is the overloading of notations. Just like in the literature, we expect to be able to denote *all* the additive laws of any ring structure by the same symbol: requiring variations for each new ring structure present in the context quickly does not scale. This is not only a parsing issue, again structure inference plays a role here.

Addressing simultaneously all these issues is a difficult task which requires taking benefit of advanced features of the implementation of the type theory of the proof assistant used. In the Coq proof assistant, the most successful recent approaches are based on type classes-like inference mechanisms [31, 32]. The SSReflect hierarchy is based on the canonical structures mechanism [31] and its implementation is described in [15]. An alternative solution based on a different type inference mechanism is described in [33]. We do not comment more here on the design of the interfaces in the SSReflect hierarchy but rather summarize its behavior and the notations we will be using throughout this article.

An exhaustive description of the whole hierarchy as implemented in the current state of the SSReflect distribution is available in the SSReflect documentation [18]. The present work stands on this existing hierarchy and extends it with ordered structures, as described in section 3. For sake of simplicity, figure 1 only describes the subset of the existing hierarchy which is actually used in the present work.

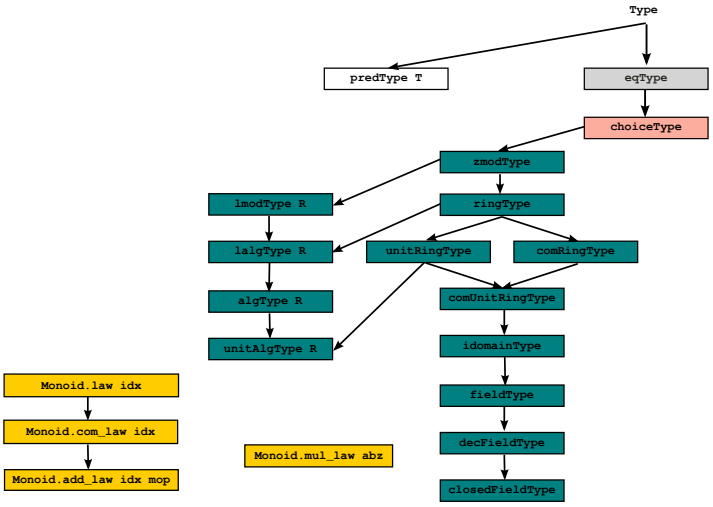


Figure 1: SSReflect algebraic structures

Each box on figure 1 represents the interface of an algebraic structure which has several implementations in the libraries. A structure is given by a carrier type, some operators on this type, and specifications for these operators and carrier. The most elementary structure is `eqType`. This structures equips a carrier type `T` with a single operator `(_ == _)` which is a binary boolean predicate, and a single specification, which ensures that this boolean comparison relation is the computable counterpart of the Coq built-in equality which is a binary predicate in sort `Prop`. An arrow between two boxes denotes an inheritance relation between the two associated interfaces.

2.2.2. *Algebraic structures.* We briefly describe on figure 2 the structures involved in the present paper and the notations they introduce. These notations will be used throughout this paper in the Coq code excerpts.

As already mentioned, all the instances of these structures are based on a type with decidable equality, denoted by `(_ == _)`. They should also be equipped with a choice operator because of the design of the present hierarchy, though this is not at all crucial to the present development.

Name of the structure	Description	Signature	Notation
`eqType`	Type with decidable equality	boolean equality test of `x` and `y`	`x == y`

`zmodType`	Commutative group	additive identity
		addition of `x` and `y`
		opposite (additive inverse) of `x`
		difference of `x` and `y`
		`n` times `x`, with `n` in nat
		opposite of `x *+ n`
		iterated sum
		`i`-th element of the sequence `l` with default value `0`
			`0`
			`x + y`
			`- x`
			`x - y`
			`x *+ n`
			`x *- n`
			`\sum_<range> e`
			`l`_i`

`ringType`	Ring	multiplicative identity
		ring image of `n`, with `n` in nat
		ring product of `x` and `y`
		iterated product
		`x` to the `n`-th power with `n` in nat
			`1`
			`n%:R`
			`x * y`
			`\prod_<range> e`
			`x ^+ n`

`unitRingType`	Ring with units	ring inverse of `x`, if x is a unit, else x
		`x` divided by `y`, i.e. `x * y^-1`
		inverse of `x ^+ n`
			`x^-1`
			`x / y`
			`x ^- n`

`lmodType R`	Left module on the scalar ring `R`	`v` scaled by `a` an element of the scalar ring	`a *: v`

Figure 2: Signatures of SSReflect algebraic structures

The `zmodType` structure of commutative group 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 [5]) 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`, `(\sum_(i \in A) F i)` sums the values of `F` belonging to a finite set `A`, `(\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 non zero ring 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 (invertible elements), commutative rings with units and integral domains. A field is a commutative ring with units in which every non zero 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.

We have however described in [12] a systematic way of constructing the required satisfiability operator from a field enriched with the property of algebraic closure, by formalizing quantifier elimination on algebraically closed fields.

2.2.3. *Instances of algebraic interfaces.* We now give a brief overview of some implementations of these interfaces we will be using in the sequel.

The Coq proof assistant provides in its core libraries an implementation of natural numbers in unary representation by defining the following inductive type with two constructors:

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

The distributed SSReflect libraries provides a library for the basic theory of arithmetic and divisibility. For the present work we needed integer arithmetic, which was not available in the distribution. We have hence developed an elementary extension to these libraries by defining the corresponding type of signed integers as:

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

This representation is unique: the `Posz` constructor builds non negative integers and where the `Negz` constructor builds negative integers: is `n` is a natural number n, `(Negz n)` represents the integer $-(n+1)$. Lifting the arithmetic operations on natural numbers to this signed version unsurprisingly leads to the definition of a `ringType` structure equipping the type `zint`. We have however not formalized a theory of signed divisibility, which would have gone beyond the prerequisites of the present work.

Polynomials provide an other important instance of the ring interface. We represent univariate polynomials as lists of coefficients with lowest degree coefficients in head position. This representation is moreover normalized by imposing that the zero polynomial is encoded as the empty list and that any non empty list of coefficients should end with a non-zero coefficient. The type `(polynomial T)` formalizes this representation of polynomials with coefficients in the type `T` as a so-called sigma type, which packages a list, and a proof that its last element is non zero:

```
Record polynomial T :=
  Polynomial {polyseq :> seq T; _ : last 1 polyseq != 0}.
```

The first `polyseq` projection of this record provides the list of coefficients of the polynomial. The `:>` symbol indicates that this projection is declared as a *coercion*, which is Coq's mechanism of explicit subtyping. Hence any inhabitant of the type `(polynomial T)` can be casted as a list of elements in `T` when needed by the automated insertion of the `polyseq` constructor. In the following, we use the notation `{poly T}` to represent the type `(polynomial T)`.

The degree of a univariate monomial is by definition its exponent. The degree of a polynomial is defined as the maximal degree of the monomials it features. A constant polynomial has degree zero, except for the zero constant which requires a specific convention: a convenient and standard choice is to set its degree at $-\infty$. To avoid introducing pervasive option types, it is convenient to replace the use of the degree of a polynomial by the one of the size of its list of coefficients. This lifts the usual codomain of degree from $\{-\infty\} \cup \mathbb{N}$ to $\mathbb{N}$ since in this case:

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

Arithmetic operations on polynomials are implemented in the expected way. From these, the SSReflect libraries declare a canonical construction of ring instance for polynomials: as soon as the type `T` is equipped with a ring structure, the type `{poly T}` inherits itself from a ring structure. Similarly, the type `{poly T}` canonically inherits from the structure of integral domain of its coefficients.

When R is an integral domain, it is no more 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 the division remains 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 $2(2X^2 + 3) = 4X^2 + 6$ by $2X + 1$ inside $\mathbb{Z}[X]$. In the context of integral domains, Euclidean division should be replaced by *pseudo-division*.

Definition 1 (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 the Euclidean division of αp by q, where α is a non null element of R which allows the Euclidean division to 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 of a polynomial `p` by the polynomial `q`, which computes `(scalp p q)` a sufficient α,

`(p %/ q)`, the corresponding pseudo-quotient, and `(p %% q)` and the corresponding pseudo-remainder. They satisfy the following specification:

```
Lemma divp_spec: forall p q, (scalp p q) * p = p %/ q * q + p %% q
```

Pseudo-remainders are extensively used in the quantifier elimination algorithm described in section 5.

Finally we also use the implementation of matrices proposed by the SSReflect libraries. Matrices are functions with a finite rectangular domain of the form $[0, m[\times[0, n[$. The type of rectangular matrices of size $m \times n$ and coefficients is denoted by `'M[R]_(m, n)` which simplifies into `'M_(m, n)` when the carrier of coefficients can be inferred from the context.

Non empty square matrices of fixed size with coefficients in a ring are canonically equipped with a structure of ring. This theory includes a definition of adjugate, cofactors, determinant, and inverse. Non empty rectangular matrices of fixed size with coefficients in a ring are canonically equipped with a structure of left module, whose internal product is denoted by `(_ *m _)` since the product of arbitrary size rectangular matrices is not a ring operation. More details on this implementation and the libraries available on matrix algebra can be found in [15, 16].

This matrix library includes syntactic facilities to define matrices by providing the general expression of its coefficients as a function of the indexes. Notations are again inspired by LaTeX-style command names. For instance the transposition operator can be defined as:

```
Definition trmx A := \matrix_(i, j) A j i.
```

Concrete examples of matrices can hence be defined by providing the enumeration of their coefficients as sequences of rows. For instance the following declaration:

```
Definition ctmat1 := \matrix_(i < 3, j < 3)
  (nth [::]
    [:: [:: 1 ; 1 ; 1 ]
      ; [:: -1 ; 1 ; 1 ]
      ; [:: 0 ; 0 ; 1 ] ] i)`_j.
```

represents the matrix:

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

with coefficients in a ring inferred from the context. The element of index (i, j) of this matrix is provided by the `j`-th element of the `i`-th element of a sequence of sequences. In order to extract the `i`-th sequence, we use the generic `nth` operation on sequences which requires a default element, here the empty sequence `[::]`. To extract the `j`-th element, we use the ring notation `l`_i` already mentioned in section 2.2.2 which hides a zero default element.

3. Ordered algebraic structures

Algebraic structures presented in section 2.2 provide a boolean operator to compare elements but no infrastructure is provided to extend this signature with an ordering relation. 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 *ordered algebraic structures*, which imposes the algebraic laws of the structure to be compatible with this order. This section is devoted to the description of lower level design choices we have adopted for this work. Some issues we address here are hence necessarily Coq specific, we have however focused our description on the solutions that could find applications in other formalizations from different areas.

3.1. **Extension of the hierarchy.** The extension of the signature of an algebraic structure with an order relation introduces a collection of elementary lemmas governing the compatibility of algebraic operations with this order, and with new operators like sign or binary extrema. The factorization of this theory as well as the existence of several widely used instances of ordered ring of fields advocates the introduction of new structures enriching the existing hierarchy, in order to benefit from the inference of unified notations and theory.

Again, we have not created a full-fledged infrastructure for binary relations, but rather plugged order theory inside the algebraic structures it is interacting with. Since we are mainly targeting real fields and more specifically number rings and fields, we have decided to work at the level of the integral domain structure. The theory of ordered group is indeed something really different from what we are studying here and we could not find any relevant example of a non integral totally ordered ring. Note that this framework does not encompass the properties of ordered semi-ring on $\mathbb{N}$, which hence still requires specific notation and theory as provided by the existing SSReflect library on natural numbers.

We extend the hierarchy described on figure 1 by introducing an ordered counterpart to the integral domain and field structures already present in the SSReflect libraries. This amounts to duplicating the corresponding branch as displayed on figure 3. The most elementary implementation we provide for the ordered integral domain interface is the type of signed integers described in section 2.2.2.

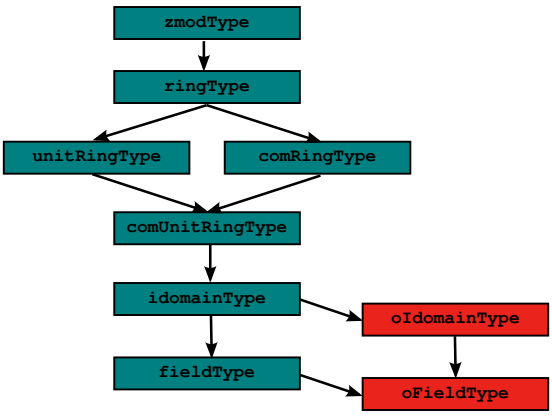


Figure 3: Extension of the hierarchy with totally ordered algebraic structures

The signature of an `oIdomainType` structure of ordered integral domain is the one of an integral domain structure, plus an extra binary relation denoted `(_ <= _)`. The specifications of this structure enrich the specifications of an integral domain with the requirements that the relation `(_ <= _)` should be a total order relation, compatible with the ring operations.

More precisely, here are the requirements that are added to the structure of integral domain in order to extend it to the structure of totally ordered integral domain:

```
forall x y, 0 <= x -> 0 <= y -> 0 <= x + y.
forall x y, 0 <= x -> 0 <= y -> 0 <= x * y.
forall x, 0 <= x -> 0 <= - x -> x = 0.
forall x y, (0 <= y - x) = (x <= y).
forall x, (0 <= x) || (0 <= - x).
```

The `oFieldType` structure of ordered field is simply the join of the structures of ordered integral domain and field. The boolean codomain of the order binary predicates imposes on purpose the validity of excluded middle on comparison statements. Throughout the paper, we call such an ordered mathematical structure with decidable comparison a discrete structure.

3.2. **Signs, case analysis based on comparisons.** The elementary theory of ordered integral domain essentially consists of numerous surgery lemmas describing how ring operations and constants combine with the order relation. We also define the binary operations of minimum, maximum and the unary operations of absolute value and sign. All these definitions are quite standard and do not deserve much comment, maybe to the exception of the sign operation. There are actually several possible choices for the type of the values of such a sign function. 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. Though rather natural, this option however does not accommodate well the common collapse of $\epsilon(x)$, the sign of x an element of an ordered ring, with 0 if x is zero and with the constant $(-1)^{\epsilon(x)}$ otherwise. 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 an ordered ring, `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 4.2). Since the ring of an element x can be different from the one we want to embed its sign $\epsilon(x)$ in, we choose to define a sign operator with integer values. Because the ring of integers is initial, there is a natural embedding of integers into any structure of ring and this solution finally proved to be the most convenient option.

A common motive in proofs involving ordered rings is a – two or three branches – case analysis according to the sign of an expression. 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 an hypothesis of the form `(x <= y)`. This is however clearly not a good option. First this does not allow for a three case analysis, second it indeed generates two subgoals, one with a new hypothesis of type `(x <= y)` and the other of type `(x <= y)= false`. In the second case, one would like at least to get directly an hypothesis of the form `(y < x)`.

This issue can be solved by working with disjunctive statements, even with sumbool types, expressing the possible results of a comparison. This approach however does not help

in the case of the three branch analysis since these disjunctions (both standard or sumbools) are binary connectives. This solution would probably require the additional support of a dedicated tactic to perform the two repeated destructions leading to the three branches.

To address this second issue, and moreover benefit from the support from the Coq unification features, we design instead specific inductive types modeling the specification of sign-based case analysis. This solution had already been proposed by G. Gonthier in the SSReflect library dedicated to natural numbers. The main 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 values of several booleans with a specification. For instance, we define the predicate `ler_xor_gtr` as:

```
Inductive ler_xor_gtr (x y : R) : bool -> bool -> Set :=
  | LerNotGtr of x <= y : ler_xor_gtr x y true false
  | GtrNotLer of y < x : ler_xor_gtr x y false true.
```

It is a binary predicate on booleans, parameterized by two elements of an ordered ring `R`. Each constructor corresponds to a propositional specification: `LerNotGtr` to the specification `(x <= y)` and `GtrNotLer` to the specification `(y < x)`. The predicate `(ler_xor_gtr x y)` relates two booleans `b1` and `b2` whenever `b1` is false (resp. `b2` is true) as soon as `(x < y)` holds and `b1` is true (resp. `b2` is false) as soon as `(y <= x)` holds. We then prove that:

```
Lemma lerP : forall x y, ler_xor_gtr x y (x <= y) (y < x).
```

This lemma establishes a rather elementary result: `(lerP x y)` is actually logically equivalent to the boolean exclusive disjunction `(x <= y) (+) (y < x)`. As expected, the proof of lemma `lerP` is almost trivial: it is mainly a case analysis on the boolean value of the comparison `(x <= y)`. The interest of the formulation of `lerP` as an inductive predicate is in fact its behavior with respect to case analysis during a proof. Indeed, the tactic:

```
case: (lerP x y).
```

performed on a goal `G` creates two subgoals, one for the proof of `(x <= y)-> G` and the other for the proof of `(y < x)-> G`. The main difference with a disjunction/sumbool-based approach is that in the statement of `G` in both subgoals, all the occurrences of `(x <= y)` and `(y < x)` have been replaced by their respective boolean values at once, and possible induced reductions have been performed accordingly. This is of special interest in the case the initial goal `G` contains `(if ... then ... else)` expressions as favored by a boolean reflection methodology. This solution also scales to the three case disjunction by defining a three constructor inductive, respectively specified by `(x < y)`, `(x = y)` and `(y < x)`.

3.3. **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.

```
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: 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 points, this approach eventually requires the use of a decision procedure for linear arithmetic. A human user is indeed soon overwhelmed by the number of constraints and unable to chain by hand the uninteresting steps of transitivity required to reach the desired condition. 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. However, our experience is that the uncontrolled growth of the context and its lack of readability remains 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.

Interval bounds are either constants or an infinity symbols. We formalize interval bounds as the following two cases inductive type parameterized by a type `T`:

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

The constructor `BClose` builds constant bounds, which are themselves inhabitants of the type `T`. This constructor takes two arguments: the value of the constant bound, and a boolean which indicates whether the extremity of the interval is open or closed. The constructor `BInfty` builds infinite bounds. Since the right or left position of the infinity symbol determines its interpretation as $+\infty$ or $-\infty$, this constructor does not need any argument. Now an interval is determined by its bounds, as modelled by the inductive type:

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

with a single constructor `Interval` taking two arguments of type `(int_bound T)`: the first one is the left bound and the second one 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 each kind of interval a predicate representing its actual characteristic function. For instance, the above interval `‘[a, b[` is interpreted as `[pred x | a <= x < b]`. At this stage, we can already rephrase the statement of our first example as the following Coq goal:

```
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 5: 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:

```
Definition int_rewrite (i : interval R) (x : R) : Prop := ...
```

which performs a case analysis on its interval argument `i` and computes the conjunction of consequences obtained from `(x \in i)`. For instance, `(int_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)`. We then prove that an interval membership assumption actually implies the corresponding conjunctions:

```
Lemma intP : forall (x : R) (i : interval R), (x \in i) -> int_rewrite i x.
```

The enhanced version of the `rewrite` tactic we use [18] 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 rewrite bases which can for instance close side conditions decided by a terminating rewrite system. The purpose of the `int_rewrite` function is to create an appropriate rewrite base gathering all the constraints we can infer from the membership to an interval.

We also provide tools to ease proofs of interval inclusion by programming a decision procedure:

```
Definition subint : interval -> interval -> bool := ...
```

which converts a problem of interval inclusion into a boolean: for instance the expression `(subint '[c, d[ ']a, b[)` evaluates to `((a < c) && (d <= b))`. We show that any interval inclusion can be proved by satisfying the boolean expression computed by the `subint` function:

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

where as presented in section 2.1, the conclusion is a notation for:

```
(subint i1 i2) -> forall x, x \in i1 -> x \in i2.
```

Now our running example in figure 5 can be solved using these facilities by the single line following command:

```
by apply: (subintP _ hx); rewrite /= (intP hc) (intP hd).
```

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

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

whose application transforms the goal into `(subint ‘[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.

4. Elementary polynomial analysis

This section 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 [2]. 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 [2] 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 [2] in a machine checked version.

4.1. **Discrete real closed fields and elementary properties.**

Definition 2. A discrete real closed field is a discrete ordered field in which the intermediate value theorem holds for polynomials.

We formalize this interface by augmenting the structure of discrete ordered field described in section 3.1 with the property of intermediate values for polynomials. Alternative presentations of real closed fields are discussed in section 7.1. In all the code excerpts of this section, we assume a type parameter `R` equipped with a structure of real closed field. The property of intermediate value for polynomials is expressed as:

```
Hypothesis ivt : forall (p : {poly R}) (a b : R),
  a <= b -> 0 \in ‘[p.[a], p.[b]] -> {x : R | x \in ‘[a, b] & root p x}.
```

where the conclusion, formalized using a Coq sigma type, is a constructive pair of the computed root and its correctness proof. This statement has many useful variants: for instance if a polynomial changes sign between two values, then it has a root between these two values. An other important consequence is Rolle's theorem:

```
Lemma rolle : forall a b p, a < b ->
  p.[a] = p.[b] -> {c | c \in ‘]a, b[ & ((p^‘()).[c] = 0)}.
```

were `p^‘()` denotes the formal derivative of a polynomial. The proof presented in [2] 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 theorems. 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 [2] (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 start 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 ordered field. Given an arbitrary non constant polynomial we define its so-called Cauchy bound as:

```
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 non zero, 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 has been formalized in a previous work by the second author [6], following the paper proof presented in [2].

4.2. **Root isolation, root neighborhoods.** In our main reference [2], 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 4.1, 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)(i : T) (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: we hence 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 2.2.3) of P is zero (which implies that P is constant), 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 = (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, then the `roots` operator is obtained by projecting this pair 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 is 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 4.1).

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 sequences of roots of a polynomial and its successive derivatives give a precise description of the behavior of a polynomial on an interval since they provide 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. An other possible default value would be to return 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. Some proofs involving neighborhoods require being able to pick a witness point in the interval they define: this is actually possible in the non degenerated cases:

```
Lemma neighpr_wit : forall (p : {poly R}) (x b : R),
  x < b -> p != 0 -> {y | y \in neighpr p x b}.
```

We now dispose of 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 non zero 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 [2] 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 should, can and will define the multiplicity of a root (see section 5.3.1) and prove that this factorization holds, we found that an induction on the size of the polynomial leads to a much more direct proofs.

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 the 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]` should be 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 both 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, the value of `(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 3.3 played an important role here to come up with an easy formalization of the easy steps of this proof. The manipulation of neighborhoods and interval 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.

5. Roots and signs

5.1. **Motivations.** The existence of a quantifier elimination algorithm for the first order theory of real closed fields can be reduced (see section 6) to the existence of a decision procedure for existential formulas of the form:

$$\exists x, (P(x) = 0) \wedge \bigwedge_{Q \in sQ} (Q(x) > 0)$$

where $P \in R[X_1, \ldots, X_n][X]$, sQ is a finite sequence of polynomials in $R[X_1, \ldots, X_n][X]$, and n an arbitrary natural number. Let us first focus on the parameter-free case:

$$\exists x, (P(x) = 0) \wedge \bigwedge_{Q \in sQ} (Q(x) > 0) \tag{5.1}$$

when $P \in R[X]$ and sQ is a finite sequence of polynomials in $R[X]$. In section 4.2, we have described how to compute the ordered exhaustive sequence of the roots of a polynomial. To solve univariate systems of sign constraints, it is sufficient to inspect the superposition of

the sequences attached to the polynomials involved in the constraints. One can even count the (possibly infinite) number of solutions. This actually provides a decision procedure for existential formulas of the form (5.1), i.e. for the case when P and elements of sQ are parameter-free polynomials in $R[X]$. This procedure however crucially relies on the computational content of the intermediate value property of the real closed field. Indeed, the sequence of roots of a polynomial is obtained by a applying the mean value theorem on intervals where the polynomial is monotonic and changes sign. Extending such a decision procedure to non closed formulas however requires further work. In the case of formula with free variables, polynomials involved in the formula are univariate polynomials in the quantified variable with coefficients themselves polynomial in the free parameters. Hence the values taken by the parameters determine the size of the polynomials, and the sign of their evaluation at a given point.

This section describes how to reconsider the problem of deciding existential formulas of the form (5.1) in order to describe a new decision procedure which scales to the non-closed case and can hence be extended to a full quantifier elimination algorithm. This amounts to expressing the decision procedure only in terms of operations reflected in the signature of real closed field, and hence independent from the presence of parameters: the decision procedure is a logical combination of sign conditions on polynomial expressions composed with the (possibly parametric) coefficients of the univariate polynomials present in the initial problem. The correctness proof of the procedure uses the intermediate value property to ensure that these sign conditions entail the existence of certain roots, but the computation process never generates a new value by a call to the intermediate value property. We first study a reduced form of the problem before extending the result to the full decision of problem (5.1). This first step is to count the number of roots of a polynomial $P \in R[X]$ on which another polynomial $Q \in R[X]$ takes positive values in a fixed bounded non-empty interval $]a, b[$.

The key ingredient of this procedure is the computation of pseudo-remainder sequences of polynomials. These remainder sequences are the core of algebraic quantifier elimination algorithms for real closed fields, like the Hörmander method [21] or the cylindrical algebraic decomposition algorithm [13, 14]. In this section, we formalize the correspondence between the signs taken by pseudo-remainders and Cauchy indexes. We then use this correspondence to count the roots of a polynomial satisfying sign conditions.

5.2. **From fields to rings.** Although the problem we study involves polynomials in $R[X]$, where R is a field, remember we want to address a further generalization to polynomials with parametric coefficients. This means that we need to use functions that do not use the inverse `(_^-1)` operation. In the proofs presented in section 4, we only used the inverse operation to compute the Cauchy bound of polynomials. Note however that this inverse operation is no more needed in the particular case of a monic polynomial. Fortunately, we can reduce problem (5.1) to the case where P is monic, which is sufficient to avoid using division (see section 5.4). This reduction to a monic polynomial is obtained by a change of variable: let p be the leading coefficient of P, if P is linear then we replace pX by X, otherwise it is also easy to prove that (5.1) is equivalent to:

$$\exists x, (S(x) = 0) \wedge \bigwedge_{R \in sR} (R(x) > 0) \tag{5.2}$$

where R is defined by $p^{|P|-2}P(X) = S(pX)$ and sR by changing sQ accordingly. It is also easy to see that S is monic.

Finally in the present section we also need to replace the Euclidean division algorithm available on $R[X]$ by the pseudo-division described in section 2.2.2. We then use the sequence obtained by iterating pseudo-division on two initial polynomials.

Definition 3. Let P and Q be two polynomials in $R[X]$. The pseudo-remainder sequence `(sremp P Q)` is a non empty finite sequence of non-zero polynomials `[::`R_0`;...;`R_N`]` defined by: $R_0 := P$, $R_1 := Q$ and $R_{i+2} := R_i$ `%%` R_{i+1}, for all $i \in \mathbb{N}$, where `(_ %% _)` denotes the pseudo-remainder defined in section 2.2.2. The sequence only contains non-zero polynomials: it is empty if P is zero.

5.3. **Pseudo-remainder sequences.** The key property of pseudo-remainders we are interested in appears when measuring the difference between the number of sign changes of a sequence of pseudo-remainder evaluated at two distinct points. The number `(var s)` of sign changes in a list of values s in an ordered field is formally defined as follows. We first compute the list of corresponding signs, skip the zeroes, and count the number of occurrences of two consecutive distinct signs:

```
Fixpoint var (s : seq R) : nat :=
  if s is a :: q then (a * head 0 q < 0) + var q else 0.
```

Note that `(a * head 0 q < 0)` is equal to 1 if `a` and `(head 0 q)` have opposite signs, and 0 otherwise thanks to the declaration of a coercion `bool >-> nat` which associates the boolean `false` (resp. `true`) with the natural number `0` (resp. `1`).

Given two points a and b and a sequence sP of polynomials, we get two lists of values by evaluating all the polynomials of the sequence respectively at a and at b. The relative number `(varp a b sP)` is the difference between the respective number of sign changes of these two lists:

```
Definition varp (a b : R) (sP : {poly R}) : zint :=
  let sPa := (map (fun P => P.[a]) sP in
  let sPb := (map (fun P => P.[b]) sP in (var sPa - var sPb).
```

The difference between the number of sign changes of the sequence of pseudo-remainder of the polynomials P and Q, evaluated at two distinct points a an b is finally computed by `(var_sremp a b P Q)`, where:

```
Definition var_sremp (a b : R) (P Q : {poly R}) : zint := varp a b (sremp P Q).
```

5.3.1. *Cauchy index.* This somehow obscure quantity `(var_sremp a b P Q)` is in fact surprisingly related to the Cauchy index of the rational fraction Q/P over the interval $]a, b[$. The Cauchy index [9] of the rational fraction Q/P at the point x is defined by:

- -1 if x is a pole and $\lim_{u \to x^-} Q/P = +\infty$ and $\lim_{u \to x^+} Q/P = -\infty$
- 1 if x is a pole and $\lim_{u \to x^-} Q/P = -\infty$ and $\lim_{u \to x^+} Q/P = +\infty$
- 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 6. 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

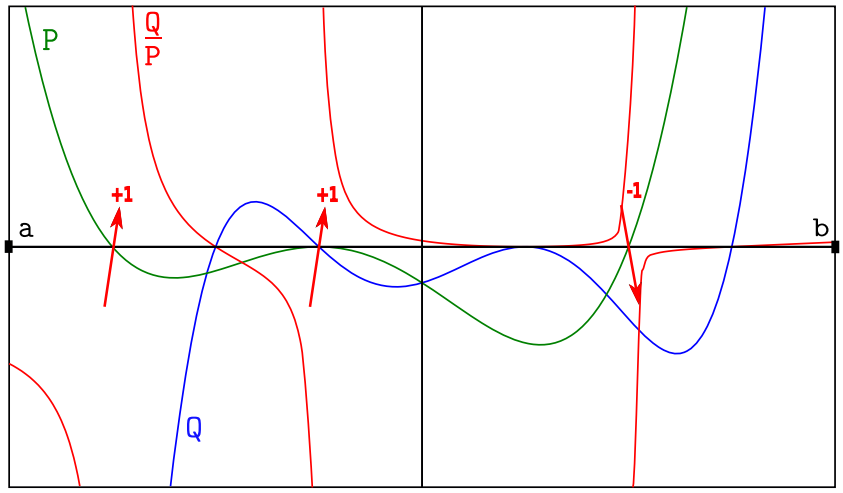


Figure 6: Cauchy index on a bounded interval

a pole is also called a *jump*. Jumps can 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, $Q/P = (X - x)^{-k}F$ 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 Q/P has a zero jump at x if and only if Q is zero or $\mu_x(P) - \mu_x(Q)$ is negative or even. If this is not the case, the sign of the jump is given by the sign of Q/P at the right of x, which is also the sign of PQ at the right of x. These remarks lead to the formalization of jump as:

```
Definition jump Q P x: zint :=
  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 `bool >-> nat` which interprets the boolean `true` as the natural number `1` and the boolean `false` as `0`. We also benefit from the definition of the sign at the right of a polynomial formalized in section 4.2. The Cauchy index of a rational fraction Q/P is formalized by summing the values taken by `jump` on the sequence of roots of the denominator P:

```
Definition cind (a b : R) (Q P : {poly R}) : zint :=
  \sum_(x <- roots P a b) jump Q P x.
```

We now prove formally that for two polynomials P and Q, as soon as a and b are not roots of any polynomial occurring in `(sremp 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, ie. that:

```
var_sremp a b P Q = cind a b Q P
```

Following the presentation of [2], the proof of this lemma goes by induction on the length of the sequence of pseudo-remainders, relying on the analogy between the property relating `cind Q P` to `(cind (P %% Q) Q)` and the property relating `(varp (sremp P Q))` to `(varp (sremp Q (P %% Q)))`. Detailing this induction was however far more technical to conduct than suggested by the reference.

5.3.2. *From Cauchy index to Tarski queries.* Recall that we consider a reduced form of our initial problem: we want to count the number of roots of a polynomial P which belong to a given interval $]a, b[$ and have a positive value when evaluated by an other polynomial Q. Formally, we want to express:

```
\sum_(x <- roots P a b) (sgr Q.[x] == 1).
```

as a combination of sign constraints on the coefficients of P and Q. The key point to solve this problem using the tools presented so far is to remark that the value of the jump of $Q \cdot P'/P$ at x is exactly the sign of $Q(x)$. But remember that the Cauchy index on a bounded interval sums the jumps of a rational fraction on the sequence of roots of its denominator, hence:

```
cind a b (Q * P^`()) P = \sum_(x <- roots P a b) (jump (Q * P^`()) P x)
```

since `(roots P a b)` contains all the poles of Q/P and a jump is zero at a point which is not a pole. If we define the *Tarski query* of a polynomial P at a sequence of points sz as the sum of the signs taken by P on the sequence:

```
Definition taq (sz : seq R) (q : {poly R}) : zint :=
  \sum_(x <- sz) (sgr q.[x]).
```

we can hence prove that the Cauchy index of $Q \cdot P'/P$ computes the Tarski query of Q on the roots of P in the bounded interval $]a, b[$:

```
Lemma taq_cind : forall a b, a < b -> forall p q,
  taq (roots p a b) q = cind a b (p^`() * q) p.
```

Since the Cauchy index can be expressed in term of signs of remainder sequences, we are almost done: we wanted to compute the expression:

```
\sum_(x <- roots P a b) (sgr Q.[x] == 1).
```

and managed to compute `(taq (roots p a b) q)` which unfolds to:

```
\sum_(x <- roots P a b) sgr Q.[x].
```

We hence need to get rid from the contribution of negative values, and satisfy the conditions on the bounds a and b. Let us postpone the discussion on the bounds, and define a generalization of Tarski queries as:

```
Definition constraints1 (sz: seq R) (Q: {poly R}) (sc : zint) : nat :=
  \sum_(x <- sz) (sgr Q.[x] == sc).
```

which counts the number of points x in the sequence sz such that $Q(x)$ has sign sz. Our reduced problem amounts to computing the value of `(constraints1 z Q 1)`.

5.3.3. *From Tarski queries to root counting.* This Tarski query of Q over z is the sum, when x ranges over the sequence of values z, of 1 when $Q(x) > 0$, of 0 when $Q(x) = 0$ and of -1 when $Q(x) < 0$. The signed integer `(taq z Q)` hence gives the number of times $Q(x)$ is positive when x ranges over z, minus the number of time $Q(x)$ is negative when x ranges over this same sequence:

```
taq z Q = \sum_(x <- z) (Q.[x] > 0) - \sum_(x <- z) (Q.[x] < 0).
```

This can be rephrased using the definitions we have introduced as:

```
taq z Q = constraints1 z Q 1 - constraints1 z Q (-1)
```

Moreover, applying the Tarski query to Q^2 and 1, we get more relations between Tarski queries and `(constraints1 z Q sc)`.

```
taq z (Q ^ 2) = constraints1 z Q 1 + constraints1 z Q (-1)
taq z 1 = constraints1 z Q 1 + constraints1 z Q (-1) + constraints1 z Q 0
```

We denote by `(tvec1 z Q)` the row vector gathering the three signed integers `(taq z Q)`, `(taq z (Q ^ 2))` and `(taq z 1)`. We denote by `(cvec1 z Q)` the row vector gathering the three natural numbers `(constraints1 z Q 1)`, `(constraints1 z Q (-1))` and `(constraints1 z Q 0)`. The relations we have stated define a 3×3 linear system:

```
Lemma tvec_cvec1 : forall z Q, tvec1 z Q = cvec1 z Q *m ctmat1.
```

where the square 3−dimensional matrix `ctmat1` is defined as follows.

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

The determinant of the matrix `ctmat1` is equal to 2, hence we can use its inverse to express `(cvec1 z Q)` in terms of `(tvec1 z Q)`. In particular `(constraints1 z Q 1)`, which is the first element of the row vector `(cvec1 z 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.

5.4. **Back to the decision problem.** The reduced problem we have solved so far is sufficient to solve the special case of our initial decision problem (5.1) when the list sQ is reduced to a singleton:

$$\exists x,\ (P(x) = 0) \wedge (Q(x) > 0)$$

Indeed, we managed to count the number of roots of the polynomial P in an interval $[a, b]$ which take positive values when evaluated by Q, provided that a and b are not root of any polynomial in a certain pseudo-remainder sequence.

Remember we have defined in section 4.1 the Cauchy bound of a polynomial and we can suppose P is monic without loss of generality. The Cauchy bound is defined only in term of the coefficients of the polynomial and provides an interval strictly containing its roots. We first compute a point b greater than the Cauchy bound of P and at which no polynomial in the sign remainder sequence `(sremp P (P^'() * Q))` cancels. Now applying

the counting algorithm on the interval $]-b, b[$ solves this special case of (5.1), since this bound is actually larger than any root of P.

In order to generalize to the case where sQ has more than one element, we first generalize the previous `constraints1` operator. The generalized version `(constraints sz sQ ssc)` checks that every polynomial in the sequence sQ satisfies the corresponding sign constraint in a sequence of sign constraints ssc, for all elements of z and we establish a relation between:

- `(taq z` $\prod_k \mathtt{Q}_k^{\varepsilon_k}$`)` with all possible $\varepsilon_k \in \{0, 1, 2\}$ for each $k \in \{1, \ldots, n\}$
- `(constraints z [::Q`$_1$`; Q`$_2$`;...; Q`$_n$`][::`σ_1`;`σ_2`;...;`σ_n`])` with all possible $\sigma_k \in \{1, -1, 0\}$ for each $k \in \{1, \ldots, n\}$

The `taq` operator remains the same as before but is now applied to products of polynomials.

There are 3^n possible Tarski query expressions, because there is a choice for ε_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 $\{1, -1, 0\}$.

We hence define `(tvec z sQ)` the row vector of all possible Tarski query expressions with z and polynomials from sQ and `(cvec z sQ)` the row vector of all possible Cauchy index expressions. If we order them properly as shown in [2], we can show that there is a linear system relating the two vectors. Yet how to obtain this linear relation is left to the reader in [2] and was a technical point of our development.

More precisely we show that

$$\forall\ \mathtt{sQ}, \forall\ \mathtt{z}, (\mathtt{tvec\ z\ sQ}) = (\mathtt{cvec\ z\ sQ}) \cdot \left(\mathtt{ctmat1}^{\otimes(\mathtt{size\ sQ})}\right)$$

where `ctmat1` is the 3 dimensional matrix seen above, $\cdot^{\otimes n}$ is the iterated tensor product n times, and `(size sQ)` is the number of elements of `z`. Note that $\mathtt{ctmat1}^{\otimes n}$ is still a unit for all n, since the tensor product of two units is still a unit. The proof is done by induction over `sQ`.

- When sQ is the empty sequence `[::]`, the iterated tensor product is the 1-dimensional identity matrix and both `(cvec z [::])` and `(tvec z [::])` evaluate to the number of elements of z.
- Otherwise, we try to prove that

  $$\forall\ \mathtt{z}, (\mathtt{tvec\ z\ (Q\ ::\ sQ)}) = (\mathtt{cvec\ z\ (Q\ ::\ sQ)}) \cdot \left(\mathtt{ctmat1}^{\otimes(\mathtt{size\ sQ}).+1}\right)$$

  assuming that

  $$\forall\ \mathtt{z}, (\mathtt{tvec\ z\ sQ}) = (\mathtt{cvec\ z\ sQ}) \cdot \left(\mathtt{ctmat1}^{\otimes(\mathtt{size\ sQ})}\right)$$

  The proofs goes by expressing `(tvec z (Q :: sQ))` using `(tvec z1 sQ)`, `(tvec z2 sQ)` and `(tvec z0 sQ)`, and also `(cvec z (Q :: sQ))` using `(cvec z1 sQ)`, `(cvec z2 sQ)` and `(cvec z0 sQ)` where
  - `z1` is the sub-sequence of `z` where we kept only elements x such that $\mathtt{Q}(x) > 0$
  - `z2` is the sub-sequence of `z` where we kept only elements x such that $\mathtt{Q}(x) < 0$
  - `z0` is the sub-sequence of `z` where we kept only elements x such that $\mathtt{Q}(x) = 0$

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 one 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 above statements, 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. We therefore 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. Theses casts are pervasive in the proofs of the general case, resulting in a considerable amount of spurious 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 approach [34] will allow for improvement in the manipulations of such datatypes.

Summary. If $(\lambda_\varepsilon)_{\varepsilon\in\{0,1,2\}^n}$ denotes the coefficients given by the first column of the inverse of $\texttt{ctmat1}^{\otimes n}$, the satisfiability of formulas (5.1) is decided by the procedure described by the expression:

$$\left(\sum_{\varepsilon\in\{0,1,2\}^n} \lambda_\varepsilon \cdot \left(\texttt{var\_sremp (-bound) bound}\quad P\left(P'\cdot\prod_{k\in\{1,\ldots,n\}} Q_k^{\varepsilon_k}\right)\right)\right) > 0$$

where `bound` is a point greater than the Cauchy bound of P at which again no polynomial in the appropriate remainder sequence cancels. This monster expression only involves comparisons between polynomial expressions in the coefficients of the polynomials featured by the atoms of the initial formula. Though this final expression is certainly unreadable by human eyes as such, programming this combination of all the elementary steps presented in this section raises no particular difficulty.

6. Quantifier elimination

We now describe how the results of the previous section are enough to provide a full quantifier elimination algorithm. The method is the same we already applied for quantifier elimination in algebraically closed fields in [12]. We here give more details and show how it adapts to the theory of real closed fields.

We first introduce notions necessary to deal with quantifier elimination in a formal way. Then we present a general transformation that applies to algorithms operating on univariate polynomials, to turn them into algorithms operating on multivariate formal polynomials.

6.1. **Deep embedding of first order logic.** A quantifier elimination algorithm is a formula transformation algorithm. We hence start by defining terms and first order formulas as objects formalized in the Coq system. We then interpret these reified terms and formulas into their shallow embedding counterparts, respectively elements of a type equipped with a field structure and first order Coq statements.

Syntax: Terms and Formulas. We assume the reader is familiar with the notion of terms and first order formulas as for instance exposed in [20]. We use an inductive type to represent terms on the signature of fields with a countable set of variables.

```
Variable R : 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 *).
```

The constructor `Var` corresponds to variables labelled with natural numbers. Any term of type `formula` built without using the `Inv` constructor can be seen as a polynomial in its variables. For example, the term `(Add (Mul (Var 0) (Var 1)) (Var 0))` corresponds to the polynomial $(x_0x_1 + x_0)$. These polynomials can as usual be considered as univariate polynomials by specializing one variable : the term
`(Add (Mul (Var 0) (Var 1)) (Var 0))` can be seen either as a polynomial in `(Var 0)` or in `(Var 1)`. Coefficients of these univariate polynomials are themselves terms, we hence define what we call formal polynomials as sequences of terms:

```
Definition polyF := seq term.
```

We also provide a function that transforms a term into a formal polynomial of the selected variable.

```
Definition abstrX (i : nat) (t : term) : polyF := ...
```

One can easily perform addition, multiplication and opposite on `polyF`. However, performing a Euclidean division is not possible, as we explain in section 6.2.

We also use an inductive type to represent formulas.

```
Inductive formula : Type :=
| Bool of bool
| Equal of term & term
| Lt of term & term
| Le 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.
```

Binders `Exists` and `Forall` are represented in named style. A quantifier free formula is represented by a term of type `formula` with no occurrences of `Exists` or `Forall`. It is easy to test whether a formula is quantifier free by a recursive inspection of its constructors:

```
Definition qf_form : formula -> bool := ...
```

Semantic: interpretation into a real closed field. We now show how this syntax is interpreted in a given real closed field `R`, provided a list of values in `R` (i.e. an environment) to instantiate free variables. In figure 7, we list the different interpretation functions we need to defined and an example of application. In the examples, we use each interpretation function with the same environment `e = [::a]`.

datatype	example	interp function	result : type
`term`	`t := Mul (Var 0)(Const 1)`	`(eval e t)`	`a * 1 : R`
`polyF`	`t := [::Var 0; Const 0; Const 1]`	`(eval_poly e t)`	`a * 'X^2 + 1 : {poly R}`
`formula`	`t := Forall (Var 0) (Lt (Mul (Var 0) (Const 1) (Var 1))`	`(holds e t)`	`forall x, x * 1 < a : Prop`
`formula` (quant. free)	`t := (Lt (Mul (Var 0) (Const 1) (Var 0))`	`(qf_eval e t)`	`a * 1 < a : bool`

Figure 7: Interpretation functions

The `holds` interpretation function builds the Coq statement corresponding to an arbitrary reified first order formula. For quantifier-free formulas, the `qf_eval` function provides an alternative, boolean, interpretation which is the truth value of the combination of atoms. The soundness of `qf_eval` is proved with respect to `holds`.

Quantifier elimination. A constructive proof of quantifier elimination consists in building an algorithm which takes a formula `(f : formula)` as input and returns a formula `(q_elim f : formula)` as output such that :

- `(q_elim f)` is quantifier free : `(qf_form (q_elim f) = true)`
- `(q_elim f)` and `f` are equivalent when interpreted in `R`:

```
Lemma q_elimP : forall (e : seq R) (f : formula),
  holds e f <-> (qf_eval e (q_elim f) = true)
```

6.2. **Full formal quantifier elimination.**

6.2.1. *From one existential to the general case.* In section 5, we described a procedure to eliminate the existential variable in a closed formula of the form:

$$\exists x, \quad P(x) = 0 \wedge \bigwedge_{i=1}^{n} Q_i(x) > 0 \tag{6.1}$$

This procedure also addresses the case of strict atoms:

$$\exists x, \bigwedge_{i=1}^{n} Q_i(x) > 0 \tag{6.2}$$

One can actually prove that a witness can be found either at $+\infty$, or at $-\infty$ or at a root of $(\Pi_{i=1}^{n} Q_i(x))'$ or that the formula does not hold. By witness at $+\infty$ (resp. $-\infty$), we mean a point greater (resp. least) than all the roots of the Q_i. Since the cases where the witness is at infinity can be expressed by a (quantifier free) sign condition on the leading coefficients of the Q_i, we can reduce the case of 6.2 to the one of 6.1.

Let us call `(dec: {poly R} -> seq {poly R} -> bool)` the decision procedure for case 6.1. We now need to explain how this can be transformed into a decision procedure on formulas with free variables $x_1, \ldots, x_{m-1}$:

$$\exists x_m, \quad P(x_1, \ldots, x_{m-1}, x_m) = 0 \wedge \bigwedge_{i=1}^{n} Q_i(x_1, \ldots, x_{m-1}, x_m) > 0 \tag{6.3}$$

Indeed, such a procedure generalizes easily to all formulas with a single prenex existential quantifier:

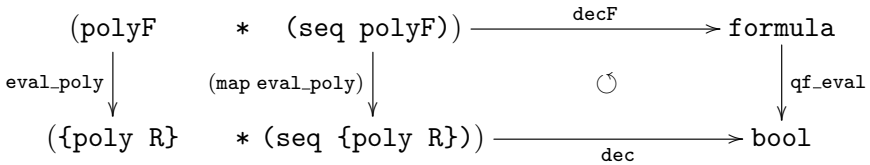
$$\exists x_m, \bigwedge_{i=1}^{n} P_i(x_1, \ldots, x_{m-1}, x_m) \bowtie_i 0 \quad \text{where} \quad \bowtie_i \in \{<, >, =\}$$

From this, it is easy to show full quantifier elimination. They key arguments are the following, see [12] for more details:

- One have to eliminate the `Inv` construction from.
- One can put the formula in disjunctive normal form.
- The treatment of a `Forall` boils down to the one of a `Exists` because atoms are decidable.

These three last steps are strictly identical to the ones we followed to prove quantifier elimination in algebraically closed fields [12].

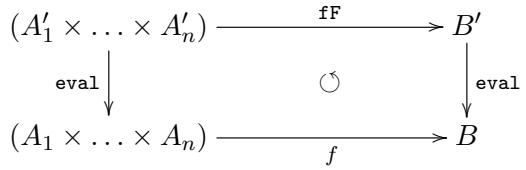
6.2.2. *Formal transformation of a procedure.* The procedure deciding (6.3) is called `decF` and has type `(polyF -> seq polyF -> formula)` : it is the formal counterpart of `dec`. It is such that the following evaluation/interpretation diagram commutes :



The arguments of the functions `decF` and `dec` are on the left hand side of the diagram. We represented them in a non-curried style on the diagrams.

The process by which we transform `dec` into `decF` is applied to all the procedures that use only operations from rings (i.e `(_ + _)`, `(_ * _)`, etc). We call DT-function a function for which this process works.

Definition 4 (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 $f : A_1 \to \ldots A_n \to B$ is a DT-function (Directly Transformable function) if we can program its reified counterpart `fF` : $A'_1 \to \ldots \to A'_n \to B'$ such that the following evaluation/interpretation diagram commutes:



Moreover, `fF` is called the direct (reified) counterpart of f

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 little examples of DT-functions for the sake of simplicity.

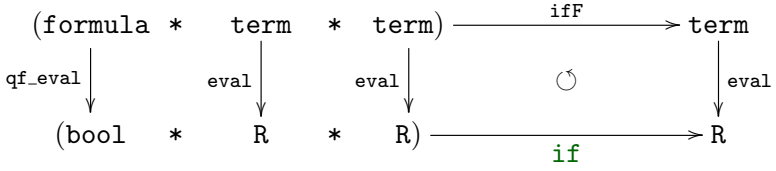
To turn a DT-function operating on values in the real closed field and on polynomials into its reified counterpart, we examine its code and turn each instruction into its formal counterpart. For example, the function `(fun x : R => x * x)` that computes the square of an element of `R` is turned into `(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 as an example the function `lcoef` that returns the leading coefficient of a polynomial:

```
Fixpoint lcoef (p : {poly R}) : R :=
  match p with
  | [::] => 0
  | a :: q => if q == 0 then a else lcoef q
  end.
```

Now let us try to turn it into its formal counterpart `lcoefF`. 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. There is no way to find a function `ifF` such that the following evaluation/interpretation diagram commutes :



As a consequence, it is impossible to find a formal counterpart of `lcoef` with type `polyF -> term`. This means that neither `if` nor `lcoef` are DT-functions. More generally, there is no direct way to find a formal counterpart to the code of an arbitrary function f that uses non DT-functions. However, it is important to notice that even if the code of a function f cannot be translated directly, it might still be a DT-function. In particular, `dec` cannot avoid using conditional statements, but in the end it will still be a DT-function.

6.2.3. *Continuation passing style transformations.* To find some reified counterparts to non DT-functions, we introduce a different reified formal counterpart to the `if` construct and more generally for any function. We call it their cps-counterpart, for continuation passing style counterpart.

The cps-counterpart to the `if` is defined as :

```
Definition if_cps (cond th el : formula) : formula :=
                                        Or (And cond th) (And (Not cond) el)
```

which requires `th` to be satisfied when `cond` is and `el` to be satisfied when `cond` is not.

With this definition, `if_cps` do 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 `decF` function, which outputs a formula.

We propose the following cps-transformation for the function `lcoef` :

```
Fixpoint lcoef_cps (p : polyF) (k : term -> formula) : formula :=
  match p with
  | [::] => k 0
  | a :: q => if_cps (q == 0) (k a) (lcoef_cps q k)
  end.
```

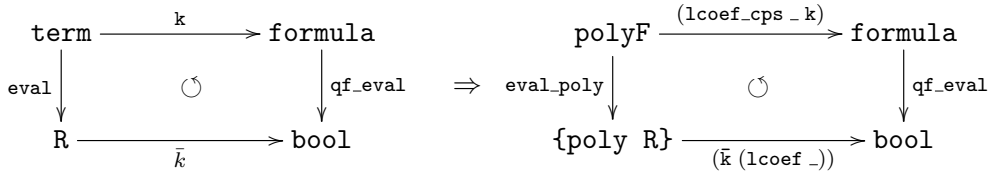
where the additional argument `k` is called a continuation.

The correctness of `lcoef_cps` with regard to `lcoef` is expressed by the following lemma.

```
Lemma lcoef_cpsP : 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 (lcoef_cps p k) = k̄ (lcoef (eval_poly e p)).
```

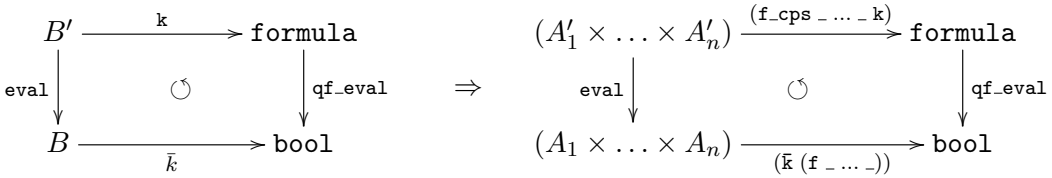
where `(k̄ : R -> bool)` is the interpretation of `(k : term -> formula)`. This lemma expresses that executing `lcoef_cps` on a polynomial `p` with continuation `k` and interpreting the result in environment `e` leads to the same result as executing `lcoef` on the interpretation of the polynomial `p` 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 `lcoef` and `k`.



More generally,

Theorem 1 (cps-counterpart)**.** Given $n+1$ types $A_1, \ldots, A_n, B$ and their formal counterparts $A'_1, \ldots, A'_n, B'$. A function $f : A_1 \to \ldots A_n \to B$ has a cps-counterpart `f_cps` : $A'_1 \to \ldots \to A'_n \to (B' \to$ `formula`$) \to$ `formula` such that the following evaluation/interpretation diagram commutes:



This means we can provide a cps-counterpart to any function, 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.

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 < lcoef p.
```

We now build the formal counterpart `testF` of test. It suffices to call `lcoef_cps` on `p` and give as a continuation the function that tests if a term is greater than 0.

```
Definition testF (p : polyF) : formula :=
                        lcoef_cps 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.

The function `dec` is also a DT-function based on various non DT-functions, including the pseudo remainder, the pseudo division, the pseudo gcd and the Cauchy bound. For each function involved in `dec`, we coded its appropriate counterpart (depending on whether it was a DT-function or not), and proved the appropriate correctness lemma.

6.3. **Decidability of the theory of real closed fields and consequences.** Quantifier elimination on a theory is well known to entail decidability of the first order formulas of this theory. This means we are able implement a Coq decision procedure for the first order theory of real closed fields. We call `sat` this decision procedure and we can use it to turn some first order formulas on a real closed field into a boolean equality. For example, if we take a Coq statement of the form :

```
forall x : R, exists y : R, F(x,y) = 0
```

where `R` is a real closed field and F(`x`,`y`) is an expression of type `R` using only operations from the field structure. Then we can replace this goal by

```
(sat (Forall 0 (Exists 1 (Equal F̄(Var 0, Var 1) (Const 0))))) = true
```

where $\bar{F}$ is the formal `term` which interpretation in `R` is F. This last goal is in fact a boolean statement (i.e. of the form `b = true`). This has a major impact on constructive

proofs because propositions from the first order theory of real closed fields can be reflected as boolean expressions.

7. Related and future work

In this section, we comment the possible extensions and applications of this formalization and comment the related work we are aware of and the limitations of ours.

7.1. **Ordered ring and real closed field structure.**

Structure of discrete real closed field. The closest work to our approach of real closed fields is the one of Robbert Krebbers and Bas Spitters in [22]. Their formalization aims at abstracting over the implementation of natural numbers and rationals in a development of Russel O'Connor in [27] about computational real numbers. Hence in particular they do not formalize general theories of ordered fields and real closed fields.

By contrast, our development addresses the properties that hold in *any* instance of the real closed field interface, like for example the decidability of its the first order theory but also the theory of polynomial functions with rational coefficients.

Using abstract interfaces, one can furthermore investigate the equivalence between different definitions of the real closed field structure. There are actually several equivalent options.

Theorem 2. Given R a totally ordered field, the three following properties are classically equivalent[1]:

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

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

(3) R is not algebraically closed, but the field $R[i]$ is algebraically closed (where i is a root of $X^2 + 1$)

In section 4.1, we made the choice to use the intermediate value theorem for the formalization, which was a convenient choice for the theory we wanted to develop since the intermediate value property was a crucial ingredient. Let us comment on the status of the equivalence stated by theorem 2 in a constructive setting. The constructive proofs that (1) $\Rightarrow$ (2), (3) $\Rightarrow$ (1) and (3) $\Rightarrow$ (2) are elementary. The missing implications require further work but it is possible to prove (2) $\Rightarrow$ (3) constructively [11]. Moreover, we believe that a formalization of (1) $\Rightarrow$ (3) can take benefit of the decidability result we have obtained in the present work for real closed fields defined using (1).

[1]There are also variants of theses properties that we do not show here.

Concrete instances of real closed fields. As already seen in section 4.1, the real closed field structure is an abstraction of *classical* real numbers, which captures the intermediate value theorem for polynomials (but not for instance the least upper bound property). Any classical axiomatization of real numbers, such as the one available in the standard distribution of the Coq system [36] can also be easily equipped with a structure of discrete real closed field as soon as the intermediate value property is formalized for at least polynomial functions.

Real algebraic numbers – i.e. real roots of polynomials of $\mathbb{Q}[X]$ – can be constructively equipped with a structure of real closed field. The first author (see [10]) has actually recently formalized a construction of the field of real algebraic numbers and proved that this construction fulfills the requirement of the interface of discrete real closed field we describe in section 4.1, and hence benefits from all the formalized theory we have presented in the previous sections. This construction furthermore demonstrates that our present formal development is not vacuous since it provides a concrete instance of the interface of discrete real closed field we designed.

The hierarchy we describe in section 2.2 requires a boolean binary function proved equivalent to the Leibniz equality on the carrier type. However, the formalized material presented in the previous section also applies when the equality relation used to prove the real closed field requirement is a decidable equivalence relation compatible with the field operations (also know as a setoid relation [1]) if the underlying type is proved to have countably many inhabitants. As a general fact, it is actually possible to construct the quotient type of a countable type by a decidable equality relation by creating a type for a collection of representatives of the equivalence classes: for instance one can choose the representant of a class to be the element with the minimal index in the enumeration of the countable carrier. More generally, the construction of this quotient type is possible as soon as the underlying type is equipped with an extensional choice operator. This quotient type itself enjoys the desired decidable Leibniz equality. The formalization of real algebraic numbers presented in [10] is actually based on such a quotient construction since in that case the construction can be realized on a type with countably many inhabitants.

However, it is not always possible or desirable to explicit a bijection between a given carrier and $\mathbb{N}$. Hence, the latter quotient type construction is not always possible. In the general case of a decidable setoid equality, it should be possible to adapt in a straightforward way all the formal proofs described in the previous section by turning the rewriting steps into setoid-rewriting steps [36], once all the unavoidable morphism proofs have been carried out. Yet in its current state, the present development is not readily available in that more general context.

Finally, when the equality relation is not decidable, whether setoid or Leibniz, quantifier elimination no more holds. If excluded middle does not hold on equality statements, one does indeed not expect the universal quantifier of the simple formula:

$$\forall x\ y,\ (x = y) \vee (x \neq y)$$

to be eliminated, otherwise this would precisely mean one can decide equalities. The quantifier elimination property vanishes similarly if the order relation is not decidable since this time one should not be able to eliminate the universal quantifier of the formula:

$$\forall x\ y,\ (x = y) \vee (x < y) \vee (y < x)$$

However, the purely existential fragment of the first order theory of a general real closed field remains decidable, by the very same proof we formalized here: deciding a purely existential

statements boils down to deciding the existence of a real root common to a given list of multivariate polynomials, which only requires solving the problem in the real closed field of real algebraic numbers. Since this remains a useful decision procedure for interesting concrete non discrete real closed field like computable real numbers, we plan to adapt our proof to make it addresses this more general case, once we come up with a Coq program executable in practice (see section 7.3).

7.2. **Remarks on the formal development.** Up to our knowledge, there is no existing formalization of real closed fields inside a proof assistant prior to the present work. However, many formalizations of real numbers have been carried out inside proof assistants. We do not cite them all, but we rather discuss and motivate the design choices we have adopted.

Polynomial fractions. The formalization of Cauchy indexes relies fundamentally on rational fractions. We believe from the presentation of [2] that a dedicated formalization was not necessary, and indeed we managed to do without. Due to the discrepancy between division and pseudo-division, it remains unclear whether a proper theory of rational fractions would have simplified our proofs.

Lack of automation. During our development, we had to solve several inequalities on a real closed field. In order to do so, we enriched our ordered rings library with several small lemmas, so that one could combine them to quickly show these goals, or modify these hypotheses. Lots of statements are so trivial that an automation procedure would be welcome to solve them automatically. However, statements which were not trivial really required the level of control that the library provides, both for understanding the proof and for transforming statements without entering manually the target statements. Moreover, with this library, trivial goals turned out to be quickly solved and did not represent critical parts of the proof.

Of course, we would be glad to diminish the "noise" caused by proofs of trivial statements, but it turns out that no existing tactic directly applied to our development. Indeed, two kind tactics could have simplified it: a decision procedure for the linear first order fragment of the theory of real closed fields and some tools for non-linear existential fragment, like sum-of-squares based techniques. Both are actually available in the Coq system [36], unfortunately their implementation is not modular enough to be adapted easily to an abstract real closed field as required by the present formalization.

Intervals. Formalization of intervals is quite independent from the implementation of reals, and can be formalized for abstract ordered fields. We compare our aim and our approach to the ones in Isabelle/HOL [26] and to the ones of Ioana Paşca [28] in Coq.

The intervals we present in this article were not meant to be the support for a development about interval arithmetic. However, it has common points with the intervals defined in [28] by Ioana Paşca. Indeed the notion of interval is reified as an inductive type and we can perform operations on them. We were essentially interested in deciding inclusion of intervals, as it is not decidable for arbitrary sets, and also in the generation of rewriting rules from an internal specification, as seen in section 3.3. We could extend our work on intervals with procedures to perform for example intersection, union (under some conditions). Apart

from the use, the difference between Paşca's formalization of intervals and ours is that we need to reflect the notion of open, closed of infinite bound.

In fact, the purpose of our intervals is comparable to the one of Isabelle/HOL. However, in the development in HOL, each lemma is associated with an equation, for each kind of interval. A same lemma is hence rewritten many times depending on whether the right bound and the left bound were open or close or infinite. When a statement involves one interval, there are nine possible cases, and up to eighty-one cases when it involves two intervals. Originally, we wrote our interval library in the same style but we were quickly overtaken by the number of cases to deal with in order to provide a complete support on the fragment we treated. As a consequence, we changed our definition of intervals to make them objects on which we could compute, but that could also be interpreted as predicates.

7.3. **Quantifier elimination as an automated procedure.** There exist different approaches for designing quantifier elimination algorithms for real closed fields in proof assistant. First, John Harrison [19] presented in his thesis a syntactic procedure for HOL Light. It is based on a rewriting system such that for each rule the left hand side is equivalent to the right hand side. Assia Mahboubi and Loïc Pottier [24] presented a procedure written in Ocaml intended to provide a tactic for Coq. This procedure was based on Hörmander algorithm, which can be found for example in [21]. Using the latter algorithm, Sean McLaughlin and John Harrison [25] also devised another proof-producing procedure for HOL Light.

Procedures defined in HOL Light are in fact tactics that build a proof of equivalence between the source formula and the target formula. The proof that it always finds a formula without quantifier and terminates cannot be expressed inside the proof assistant, but as a meta-theoretical result. Of course the procedure is correct because it uses only primitives from the system, but there is no formal proof that the procedure is complete.

Unlike the last procedure from S. McLaughlin and J. Harrison, our procedure is formally proved correct and complete, but is totally ineffective for the time being. The datastructures we adopted for the formalization are indeed quite naive from an algorithmic viewpoint. Moreover, in the formal definitions we use for basic operations like polynomial and number arithmetics, reduction is blocked on purpose to avoid unwanted behaviors during the proofs. While the latter issue is rather easy to speed, we believe that significant speed improvement might best be obtained by using a sparse representation for polynomials, and efficient algorithms for computing Euclidean division and gcd. Our experience is that the datastructures adapted to the formal mathematical proof of correctness and the ones adapted to efficient computations have little chance to coincide. Hence we suggest to split the formal proof of correctness of efficient algorithms on efficient datastructure into two parts: first the mathematical correctness result, based on naive datatypes, and then the proofs that optimized algorithms and representations are correct with respect to the ideal, mathematical ones.

Since the present paper describes the first step of such an approach, the main contribution of the present work is a theoretical decidability result more that a proof-producing automated decision procedure. However, considering the intrinsic complexity of the algorithm we have proved correct so far, we will likely not complete the second part nor push the formalization to make it executable. We plan instead to reuse the tools described here to prove the correctness of the Cylindrical Algebraic Decomposition, which is far more efficient in theory. This procedure has already been programmed in Coq, by Assia Mahboubi [23], but the proof is still incomplete.

Acknowledgments

The authors wish to thank Georges Gonthier for his precious suggestion to use continuation-passing style in the last part of this work. The authors are also greatly indebted to two anonymous referees for their valuable comments and suggestions on a previous draft which lead to significant improvements of this work and of its presentation.

References

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

[2] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag New York, Inc., 2006.

[3] 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.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[5] 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.

[6] Yves Bertot, Fréderique Guilhot, and Assia Mahboubi. A formal study of bernstein and coefficients polynomials. *Mathematical Structures in Computer Sciences*, 2011.

[7] Jack Bochnak, Michel Coste, and Marie-Françoise Roy. *Real Algebraic Geometry*, volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1998.

[8] 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.

[9] Auguste Cauchy. Calcul des indices des fonctions. *Journal de l'Ecole Polytechnique*, 15(25):176–229, 1832.

[10] Cyril Cohen. Construction des nombres algébriques en Coq. In *Proceedings of JFLA2012 (to appear)*, 2012. `http://perso.crans.org/cohen/work/realalg/`.

[11] 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`.

[12] 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.

[13] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition–preliminary report. *SIGSAM Bull.*, 8:80–90, August 1974.

[14] George E. Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *J. Symb. Comput.*, 12:299–328, September 1991.

[15] 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.

[16] Georges Gonthier. Point-free, set-free concrete linear algebra. In *Interactive Theorem Proving, ITP 2011 Proceedings*, Lecture Notes in Computer Sciences. Springer, 2011. to appear.

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

[18] 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`.

[19] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[20] Wilfried Hodges. *A shorter model theory*. Cambridge University Press, 1997.

[21] Lars Hörmander. *The analysis of linear partial differential operators*, volume 2. Springer-Verlag, Berlin etc., 1983.

[22] 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. to appear.

[23] Assia Mahboubi. Implementing the cylindrical algebraic decomposition within the coq system. *Mathematical Structures in Computer Science*, 17(01):99–127, 2007.

[24] Assia Mahboubi and Loïc Pottier. Élimination des quantificateurs sur les réels en Coq. In *Journées Francophones des Langages Applicatifs, Anglet*, January 2002.

[25] Sean McLaughlin and John Harrison. A proof-producing decision procedure for real arithmetic. In Robert Nieuwenhuis, editor, *CADE-20: 20th International Conference on Automated Deduction, proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 295–314, Tallinn, Estonia, 2005. Springer-Verlag.

[26] 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`.

[27] Russell O'Connor. *Incompleteness & Completeness, Formalizing Logic and Analysis in Type Theory*. PhD thesis, Radboud University Nijmegen, Netherlands, 2009.

[28] 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.

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

[30] The Mathematical Components Project. SSReflect extension and libraries. `http://www.msr-inria.inria.fr/Projects/math-components/index_html`.

[31] Amokrane Saibi. Typing algorithm in type theory with inheritance. In *Principles of Programming Languages, POPL 1997 proceedings*, pages 292–301, 1997.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] The Coq Development Team. *The Coq Proof Assistant, Reference Manual*. `http://coq.inria.fr`.