Formalizing real analysis for polynomials

Cyril Cohen

Formalizing real analysis for polynomials

Cyril Cohen
INRIA Saclay – Île-de-France
LIX École Polytechnique
INRIA Microsoft Research Joint Centre
Cyril.Cohen@inria.fr

December 12, 2010

Abstract

When reasoning formally with polynomials over real numbers, or more generally real closed fields, we need to be able to manipulate easily statements featuring an order relation, either in their conditions or in their conclusion. For instance, we need to state the intermediate value theorem and the mean value theorem and we need tools to ease both their proof and their further use. For that purpose we propose a Coq library for ordered integral domains and ordered fields with decidable comparison. In this paper we present the design choices of this libraries, and show how it has been used as a basis for developing a fare amount of basic real algebraic geometry.

1 Introduction

The purpose of this work is to show how to formalize some real analysis. We particularly would like to point out how important are the choices we make in stating single lemmas. We also describe the basis we could have chosen to built the theory we describe, and why we think we made a good choice.

In order to achieve this goal, we reuse techniques from the mathematical components team to build a little bit of analysis, but only with polynomials for now. This allows to use only algebraic properties to express and prove analytic theorems. We hope these techniques ease the formalisation and the manipulations inside a Type Theoretic proof assistant such as Coq. And Coq's system is based on an intuitionistic logic.

When working in such an intuitionistic context, we don't have the excluded middle axiom anymore. However, this axiom is extensively used in standard mathematical reasoning and mathematical theorems are usually proved without wondering whether their proof is constructive or not.

A first solution would be to add the excluded middle to Coq. But this has two major drawbacks. The first one is that Coq is also a programming language, and supposing excluded middle is like supposing the existence of a program which doesn't exist. The second one is that it could introduce inconsistencies in the system, enabling us to prove the false, and thus prove anything.

An alternative, proposed by the SSReflect extension for Coq, is to have classical reasoning with a better granularity. Indeed, instead of supposing the excluded middle true for any property, we work with structures that provide a restricted version of excluded middle. For example, we define a structure of *discrete type* by requiring it to have decidable equality (this is an excluded middle limited to equality predicate on a given type).

This means that if we can find such a type, we just see it as an instance of this structure. For example, equality on natural number is decidable and thus we can view natural numbers as an instance of *discrete type*). This means that this restricted kind of excluded middle can be instantiated by the decision procedure of equality on the given type. Moreover, this introduces no inconsistency in the system because we never supposed there were an instance of the structure.

A third advantage is that we can represent decidable propositions as booleans. Since equality on boolean is decidable, the proof that two given booleans are equal is unique. This implies we can prove equivalence between two boolean predicate by proving the equality of these two predicate. That's why we use boolean predicates as much as we can, instead of propositions.

These techniques come historically from the proof of the *four color theorem* (cf [6]), which relies on combinatorics. This is a domain where boolean reflexion proved very useful.

The first part of this paper will describe what are the basis we took for our development. Then we discuss the importance of the way we stated lemmas, from a user point of view, and how we should consider it from a developing point of view. In a third part, we show how far we can go into mathematics of the theory we deal with. Finally we show an overview of some work in progress we derive from the latter theory.

We here do not pretend to build a notion of topology nor covering the basis for entire analysis. In fact, we intend to prove specific mathematics results, but we try to remain generic with regard to the way we enunciate notions and theorems. We then still have some hope the notions could be generalized to a bit more analysis, keeping the same kind of light formulation.

Disambiguation of the meaning of *axiom* When talking about *axioms*, one has to be careful whether he talks about an *axiom* of the system **in which** he is working (for example Coq), or an *axiom* of the theory **on which** he is working (for example, the axioms of the theory of groups, or of *real closed field*). We will call *global axiom* the axioms of our system (which correspond to the `Axiom` command in Coq), and use *axiom* for the properties that generate some theory.

2 A representation for discrete *real closed fields*

Real analysis requires fixing a representation of real numbers. However, from a constructive perspective real numbers are not an easy topic. There are multiple possible implementations of $\mathbb{R}$.

2.1 Implementation of real numbers

We discuss here the troubles we may encounter with real numbers, depending on the implementation we choose.

Classical Real numbers These are the real numbers we use in mathematics. They could be classically built as the completion of $\mathbb{Q}$.

They are currently integrated in Coq by several *global axioms* (cf Coq standard library *Coq.Reals.Raxioms*). These *global axioms* give us the decidability and the totality of the order, archimedean property and the completeness property.

They are nevertheless pretty unsatisfactory because they have no implementation in Coq. No one could provide axiom-free Coq proof terms that satisfies these. No one could rely on them to compute or extract a program. It could even be considered as unsafe because when we load these real numbers into the system, we add definitively new *global axioms*. This endangers the consistency of the system. Indeed, adding *global axioms* incompatible with the system or incompatible between themselves leads to the ability to prove `False`.

Constructive Real Numbers There is also a constructive point of view of reals, as done in *CoRN*. Real numbers are here represented by functions outputting an approximation of the number. This has the good property to be constructible and hence have a concrete implementation. The construction could be made out of Cauchy sequences among other ways, as explained in [5].

There is even a work (cf [8]) which relates the axiomatizes real numbers with the constructive one from *CoRN*, in a specific context. However, equality is not decidable in general. Hence ordering is not decidable either. Thus, we are not in the framework we chose.

Algebraic Real Numbers The most satisfying model one for our purpose are *algebraic real numbers*. They are real roots of polynomials with coefficients in $\mathbb{Q}$. They are a subset of constructive real numbers and have stronger properties. They still are contructive but they have also decidable equality, and also decidable order.

These kind of real numbers seem to fit our requirements. So we may decide to base our development onto them. However, no implementation of *algebraic real numbers* has been, to our best knowledge, done in Coq.

2.2 An interface for real numbers

However, we don't really need an implementation for our purpose. We are only interested in the first order theory of *algebraic real numbers*. That's why we formalize the theory of *discrete real closed fields*. This mean we only have to specify a structure requiring the *real closed fields* axioms. Then, the result we prove is valid for any model of *real closed fields*, including an implementation of *real algebraic numbers*. Moreover, giving such an implementation will validate our results, which all have the existence of *real algebraic numbers* as an hypothesis.

A *real closed field* is an *ordered field* where the *intermediate value theorem* holds for polynomials. We thus have to dispose of a representation of (*discrete*)

ordered fields or more generally (*discrete*) *ordered integral domains*. Indeed, this allows to cover the case of relative numbers, which are a model of *discrete ordered integral domains*. And also rational numbers, which are a model of *discrete ordered fields*.

We then add the *intermediate value theorem* (abbrev. *IVT*) as an axiom to get the theory of *real closed field*. This enables the possibility to do analysis. The *IVT* states that if a polynomial P changes its sign between a and b, then it has a root between a and b. We see in Section 4 that all this allows us to prove basic real analysis results, for polynomials. This is the least we could expect from this structure in order to do analysis.

3 About implementation techniques

3.1 Ordering and *real closed fields* in Coq

SSReflect libraries features an algebraic hierarchy (defining Rings, Integral Domains, and Fields for example). This hierarchy contains everything we need to formalize *real closed fields* except for the part which deals with ordering. Thus, we extend this hierarchy by adding an order structure on it. To build the hierarchy, the choice for representing the abstractions of Rings and Integral Domains (etc ...) has been to use Canonical Structures and Coercions (cf [4]).

The algebraic hierarchy has been built out of components called *classes*. A *class* is simply a record containing the laws and properties of a given structure. A class is made of what we call *mixins*. *Mixins* are the laws and properties we must add to a class, to make it a sub-class. For example the class of a *fields*, is made of the class of *integral domains* plus a mixin telling that units are the non null elements.

Mixin for Ordering Since we want our order to be decidable, we choose `le` to be a boolean predicate, this choice is justified in 3.2/ The order relation symbol is the only one required since we can deduce the strict inequality `lt` from the large one `le` and equality. The additional axioms for ordering say that `le` has to be a antisymmetric, transitive and total, and must be compatible with `+` and `*`.

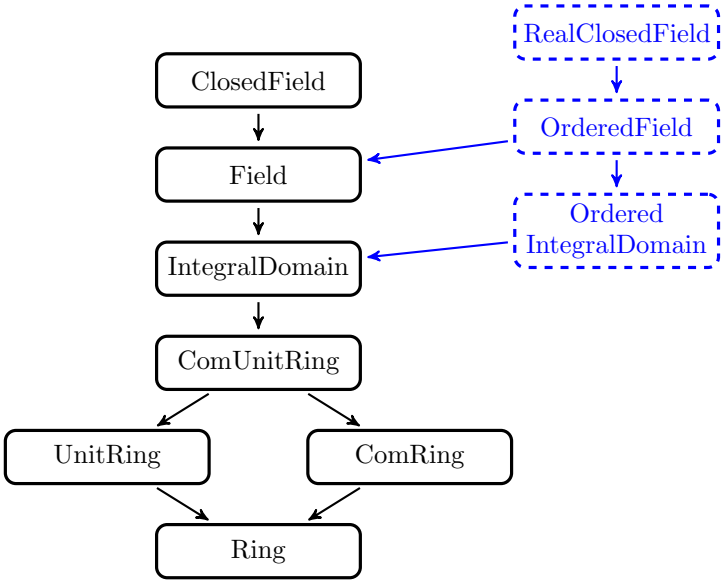
Here is the order *mixin* we chose.

```
Record mixin_of (R : ringType) := Mixin {
  le : R -> R -> bool;
  _ : antisymmetric le;
  _ : transitive le;
  _ : total le;
  _ : forall z x y, le x y -> le (x + z) (y + z);
  _ : forall x y, le 0 x -> le 0 y -> le 0 (x * y)
}.
```

with a notation `x <= y` for `le` and `x < y` for `lt`.

Integration in algebraic hierarchy We have now to decide where to add this *mixin* in the existing algebraic hierarchy. Since we have no example of *ordered ring* below integral domains, and the *relative numbers* are an *ordered*

integral domain, we chose to extend the hierarchy with an *ordered integral domain* structure, an *ordered field* structure and a *real closed field* structure:



3.2 Pragmatic formalization of lemmas

While building a library, the way functions and lemma are stated is as important as the content they provide. The key point is that the objects of the library should be as easy as possible to use. Choosing the way to state a lemma should depend on the advantages it provides.

We try here to make a list of principles we used and that makes life easier for the user. These principles are folklore for SSReflect advanced users but are not necessarily obvious. We illustrate these principles on examples either from SSReflect library or from our development about *ordered rings*.

Boolean reflexion Since we try to use as much as boolean predicates as we can to represent proposition, we need tools to ease the use of such lemmas. First we would really want that boolean properties make sense as statements. Indeed if `eqT : T -> T -> bool` is a decision procedure for some equality over the type `T`, we would like to state properties like that :

```
Lemma eqT_correct : forall x y, (x = y) <-> (eqT x y).
```

But this makes no sense as such because `eqT x y` has type `bool`, not `Prop`. However, using Coq coercion system, we can define the following function as an implicit coercion.

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

This enables us to state the previous lemma, which is in fact the following one, but where *is_true* has been automatically inferred, and hidden at pretty printing.

```
Lemma eqT_correct : forall x y, (x = y) <-> is_true (eqT x y).
```

The example `eqT_correct` we showed is what we call a reflexion lemma. This kind of lemma relates the boolean construction with an equivalent propositional formulation. And there are facilities to go from one representation to the other. The purpose of the equivalent proposition formulation is to provide new functionalities. For example, rewriting with `e : eqT x y` rewrites subterms `eqT x y` to `true`, whereas `eqT_correct e` rewrites `x` to `y`.

Privileging rewriting As a side effect and feature of the previous principle, equality is used instead of equivalence as much as possible and hence rewriting becomes one of the most used tactic in SSReflect.

Applied to our development, rewriting with a boolean property `exy : x <= y` which is in fact `exy : x <= y = true` replaces every occurrences of `x <= y` by true, in any context. This is easier than trying to isolate the predicate using a lot of case analysis and constructor introductions to try to apply it at the right places. This also enables us to use equivalence lemmas stated as equalities, such as `~~(x <= y)= (y < x)` and replace occurrences of `~~(x <= y)` by `y < x` (where `~~` is the boolean `not`).

Combining rewrites The SSReflect rewrite tactic has a different behavior than Coq standard one. We can give it pairs of lemmas to rewrite with. For example, if we have `lxy : x <= y` and `lyz : y <= z`, we can write `rewrite (lxy, lyz)`.. The term `(lxy, lyz)` has type `(x <= y)* (y <= z)`. Rewriting with this product will try to rewrite with the first element of the pair, and if it fails, the second one.

We used this feature to encode intervals. To state that x is between a and b, we can write `a <= x <= b`, which is a notation for `a <= x && x <= b`. This has the nice property to be a boolean, so we can combine it with boolean operators and equate it with another boolean to state some equivalence. However, once we have it in our hypotheses, it becomes less useful. Imagine we have the following goal.

```
a b x : R
x_in_ab : a <= x <= b
=====================
a <= x
```

This goal seems trivial, however, to solve it, we still have to translate the boolean conjunction into a propositional one, split it and rewrite with the left part. Now, if we state intervals in this way : `x_in_ab : (a <= x)* (x <= b)`, thanks to pair rewriting, we just have to rewrite with `x_in_ab`. Moreover, we can define the following reflexion lemma :

```
Lemma ccintP : (a <= x) * (x <= b) <-> a <= x <= b.
```

And in the previous goal

```
a b x : R
x_in_ab : a <= x <= b
=====================
a <= x
```

rewriting with `(ccintP x_in_ab)` turns the goal to `true`.

We can also chain rewriting, by `rewrite lemma_1 lemma_2...lemma_n`.

Conditional rewriting When we rewrite with a lemma such as

```
Lemma ltrW : forall (x y : R), x < y -> x <= y
```

, if it successes, it will generate a new subgoal, asking for us to show `x < y`.

For example, with the following goal

```
a b : R
lab : a < b
======
a <= b
```

doing a `rewrite ltrW` leads to two subgoals.

```
a b : R
lab : a < b
====
true
=====
a < b
```

But the `rewrite` command has a special syntax `//` to eliminate trivial subgoals. And in the previous example, both are trivial. Thus, `rewrite ltrW //` will solve the entire goal.

However, if the subgoals are more complicated, `//` doesn't work anymore. For example with the following goal.

```
a b c: R
lac : a < c
lcb : c < b
==================
(a <= b) && (a + c < 2 * b)
```

`rewrite ltrW` as well as `rewrite ltrW //` will both lead to the two subgoals :

```
a b c: R
lac : a < c
lcb : c < b
======================
true && (a + c < 2 * b)
=====
a < b
```

Hopefully, the `?` modifier allows to try to rewrite.
For example if we do `rewrite ltrW ?(ltr_trans lac)//` (where `ltr_trans` is the transitivity lemma for `<`) , we get only one subgoal :

```
a b c: R
lac : a < c
lcb : c < b
======================
true && (a + c < 2 * b)
```

Factorizing Lemmas and Naming conventions Due to these principles we end up with a large amount of lemmas. For example in our development about *ordered rings*, we have actually about 140 lemmas available to the user.

However, there could have been a lot more if we didn't use the previous and the following principles. Nevertheless, 140 is still huge, and we really don't want the user to get lost in the labyrinth of lemmas of the library. A good way to help him is to adopt systematic naming convention. This idea comes also from the *four color theorem formalization* (cf[7]).

The basis of this convention is naming the lemma by the defined symbols that appear in its statement, generally in order of appearance. For example:

```
Lemma sgr_mul : forall x y, sgr (x * y) = sgr x * sgr y.
```

is named as such because it involves `sgr` and `mul`.

We can put suffixes to add some more information when needed. For example, in

```
Lemma ler_add2r : forall z x y, (x + z <= y + z) = (x <= y).
Lemma ler_add2l : forall z x y, (z + x <= z + y) = (x <= y).
```

the suffix `2r` (resp. `2l`) means that we add on both sides of the inequality (`2`) and on the right side of the addition (`r`) (resp. on the left side of the addition (`l`)).

Having these systematic naming conventions have two major advantages. The first one is that user in the know can almost guess the statement associated to the lemma without having to `Check` or `Print` it. The second one is that a user looking for a lemma that talks about some defined symbols only has to do a `Search` on these names to find quickly the right one. We think this methodology has been so far very efficient.

Positioning arguments correctly When we write a lemma, we could place the arguments in almost any order we want as long as it respects dependencies. Whereas it doesn't matter in a mathematical point of view, it matters while concretely using the lemma in another proof. We must place in the first positions what the user needs to provide the most.

For example, the following

```
Lemma ler_add2r : forall (x y z : R), (x + z <= y + z) = (x <= y)
```

should be rewritten this way :

```
Lemma ltr_add2r : forall (z x y : R), (x + z <= y + z) = (x <= y)
```

because if we want to rewrite from left to right, we would like to be able to explicit `z` without giving explicitly additional arguments (or put `_`).

3.3 Destructing with a specification

For case analysis over the trichotomy ($x < y$ or $x = y$ or $x \geq y$) we use a feature of the `destruct` Coq tactic. This works by building the following inductive

```
CoInductive ltrgt_spec x y : bool -> bool -> bool -> Set :=
  | LtrgtLt of x < y : ltrgt_spec x y true false false
  | LtrgtGt of x > y : ltrgt_spec x y false true false
  | LtrgtEq of x = y : ltrgt_spec x y false false true.
```

along with a lemma

```
Lemma ltrgtP : forall x y, ltrgt_spec x y (x < y) (x > y) (x == y
    ).
```

Then, destructing `ltrgtP` automatically produces three subgoal with the corresponding hypothesis on top of the stack. Moreover, this destruction replaces occurrences of `x < y`, `x > y` and `x == y`.

For example while proving

```
Lemma mulf_gt0 : forall x y, (0 < x * y) = ((0 < x && 0 < y) || (
    x < 0 && y < 0)).
```

Once we introduce `x` and `y`, we get the subgoal

```
x y : R
============================================
(0 < x * y) = ((0 < x && 0 < y) || (x < 0 && y < 0))
```

If we do `case: (ltrgtP 0 y)` we get three following subgoals. What's interesting here is to see how much work it did in only one `case` analysis : it generated three subgoals with a usable hypothesis on the left, and it substituted `0 < y`, `y < 0` and `y == 0` by `true` or `false`, depending on the case.

```
x y : R
============================================================
0 < y -> (0 < x * y) = ((0 < x && true) || (x < 0 && false))

============================================================
y < 0 -> (0 < x * y) = ((0 < x && false) || (x < 0 && true))

============================================================
y = 0 -> (0 < x * y) = ((0 < x && false) || (x < 0 && false))
```

3.4 An attempt to a little more automation

In an attempt to decrease the number of lemma, we tried to regroup some lemmas that could talk both about `<=` and `<`. For example, we could make one lemma gathering the two following results :

```
Lemma ler_add2r : forall (z x y : R), (x + z <= y + z) = (x <= y)
    .
Lemma ltr_add2r : forall (z x y : R), (x + z < y + z) = (x < y).
```

into this only one lemma :

```
Lemma lter_add2r : forall (c : lter R) (z x y : R),
                       c (x + z) (y + z) = c x y.
```

Where `lter R` is a structure capturing both `<` and `<=`. the good `c` is then inferred while rewriting.

After using it a little, we discovered it was not as easy to use as it seemed to be in the beginning. Indeed, rewriting with such lemmas allowed to factorize a little more. But with undesirable side effects, both in understanding and in practicing. The lemmas are more difficult to understand, and rewriting with them leads to a term with a beta-redex that we'd never want to see.

This design choice will be removed while updating this part of the *order extension*.

4 Into the basic theory of *real closed fields*

Basing ourselves on the theory of *real closed fields*, we'd like to show we can derive the basic facts about real analysis for polynomials.

4.1 *Rolle*'s theorem and *Mean Value theorem*

The first thing is that we can quite easily prove constructively Rolle's theorem for polynomials which is a basic theorem in real analysis.

$$\forall a, b \in R, p \in R[X], a \leq b \wedge p(a) = p(b) = 0 \Rightarrow \exists c, a < c < b \wedge p(b) - p(a) = p'(c) \cdot (b - a)$$

And the Mean Value Theorem, which is a corollary to Rolle's Theorem. It is five lines of SSReflect proof script.

$$\forall a, b \in R, p \in R[X], a \leq b \Rightarrow \exists c, a < c < b \wedge p(b) - p(a) = p'(c) \cdot (b - a)$$

A constructive proof of Rolle's theorem for polynomials The statement of *Rolle's theorem* in SSReflect is

```
Lemma rolle : forall a b p, a < b ->
      p.[a] = 0 -> p.[b] = 0 ->
  exists c, a < c < b /\ p^`().[c] = 0.
```

(where `p.[a]` is the evaluation of a polynomial `p` and `p^()`` is its derivative).

But to prove it, we had to prove an intermediate lemma:

```
Lemma rolle_weak : forall a b p, a < b ->
      p.[a] = 0 -> p.[b] = 0 ->
  exists c, a < c < b /\ (p^`().[c] = 0 \/ p.[c] = 0).
```

which states that we can find c where either the derivative is null, or the polynomial itself is. Fortunately, we can prove that any polynomial has a number of root lower that its degree. So we can iterate the previous lemma a finite number of times (bounded by the degree), to prove *Rolle's theorem*.

Sketch of the proof of `rolle_week` We factorise P into $P = (X-a)^{n+1}(X-b)^{m+1}Q$

Then there are two cases :

- Either $Q(a)Q(b) \leq 0$, and by IVT, we can find c such that $a < c < b$ and $Q(c) = 0$.

  We conclude that $P(c) = 0$

- Or $Q(a)Q(b) > 0$.

  We have $P' = (X-a)^n(X-b)^m R$ with

  $$R = (X-a)(X-b)Q' + (m+1)(X-a)Q + (n+1)(X-b)Q$$

  Then we notice that $R(a)R(b) < 0$.

  Hence there exists c such that $a < c < b$ and $R(c) = 0$. Thus $P'(c) = 0$

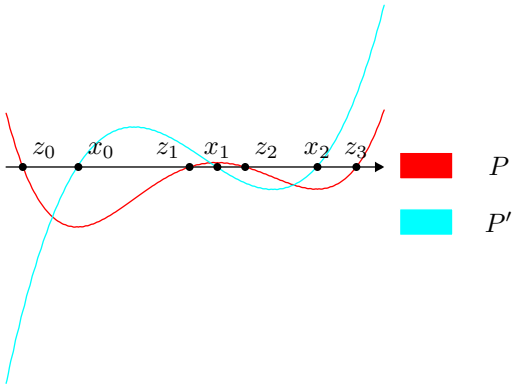
4.2 Root isolation

Given a and b (such that $a < b$), we are also able to compute the list of the roots of a non zero polynomial, that are between a and b (excluded).

The method is the following, we proceed by induction on the degree of the polynomial. Since P is non null, the base case is $\deg(P) = 0$. In this case, there are no roots, we return the empty list.

The induction hypothesis is "for all P such that $\deg(P) \leq n$, we can compute the roots of P". Let's take P such that $\deg(P) = n + 1$, we have to show that we can compute its roots.

We can apply the induction hypothesis to P' whose degree is $n' < n$, and get the list $x_1, \ldots, x_m$ of the roots of P'. Let's pose $x_0 = a$ and $x_{m+1} = b$. For all $0 \leq i \leq m$, P' has the same sign between x_i and x_{i+1}, hence P is monotonic on $[x_i, x_{i+1}]$. This means it has a root on $[x_i, x_{i+1}]$ iff $P(x_i)$ and $P(x_{i+1})$ have different signs (by IVT and monotonicity). Moreover, if it has one, it's the only one (by monotonicity). Let z_i be the root if it exists. the z_i for i such that z_i is well-defined plus the x_i that are also roots of P are the roots of P. Qed.



5 Works in progress

Root isolation allows to define a constructive notion of neighbourhood for polynomial properties.

5.1 Neighborhood reasonning

At some point we'd like to reason about what happens near a point. This notion isn't straightforward to formalize because mathematician use sometimes the expression "near enough" without taking care to mention with regard to what. And even if they say $\forall\varepsilon, \exists\eta(\varepsilon), s.t. P(\varepsilon, \eta(\varepsilon))$, this doesn't mean that $\eta : \mathbb{R}^+ \to \mathbb{R}^+$ is computable. This seems to prevent us from building an equivalent.

While this may be true in the general case, in some cases where the η function is computable, we'd like to be able to handle it. For example, if the property that characterizes the neighborhood is that some polynomial P doesn't have a root, since we can isolate roots, we can explicit things.

Here is an example where such reasoning is needed

Sign on the right of root Let $\mathrm{sgp}_{\mathrm{right}}(P, x)$ be a function that gives the sign of a function in the right neighborhood of x. This means $\mathrm{sgp}_{\mathrm{right}}(P, x)$ is the sign of $P(y)$ on any y between x and the first root of P from x (excluded).

There is also an algebraic characterisation of $\mathrm{sgp}_{\mathrm{right}}$. Indeed, $\mathrm{sgp}_{\mathrm{right}}(x)$ is the sign of $P^{(n)}(x)$ for the first natural number n such that $P^n(x) \neq 0$. So the following program computes this sign.

```
Fixpoint sgp_right (p : {poly R}) x :=
  if p.[x] != 0 then sign p.[x] else sgp_right p^`()
```

And at some point we need to relate the specification we required at first with the definition we finally stated. We would like a lemma of this form

$$\exists \eta_x, \forall y, x < y < x + \eta_x \Rightarrow \mathrm{sgp}_{\mathrm{right}}(x) = \mathrm{sign}(P(y))$$

.

To prove this constructively, we need to build a good η_x. Thanks to the root isolation procedure we can find such an η_x. For example, $\eta_x = x' - x$ where x' is the first root of P between x and b for any b such that $x < b$. (Thus b could be $x+1$ as well as the upper bound of an interval on which we may be working).

Some experimentations showed that proofs seemed to be easier if we leave the choice of b to the user. This makes an $\eta_{x,b}$ dependent not only from x but also from b. Moreover, we decided to turn $x < y < x+\eta_{x,b}$ into a predicate on y, to make statements more readable. This corresponds to the right neighborhood of x in $]x, b[$.

Defining neighborhood predicates This leads us to define a neighborhood predicate, rather than defining a bound ($\eta_{x,b}$). This neighborhood predicate's graph is the set of y that are close enough of x on the right. Close enough in the sense their image by P all have the same sign.

We also need the same reasoning on the left. Here are the definitions we finally posed.

```
Definition neighpr p x b := (fun y => x < y < head b (roots_of p
    x b)).
Definition neighpl p a x := (fun y => last a (roots_of p a x) < y
     < x).
```

Since this part of the work is still in progress, this is subject to change. But since the rest of the development (cf 5.2) that already uses these notion is almost done. But we can already guess they are a good abstraction over neighborhood for our particular purpose.

5.2 Towards quantifier elimination

As an application of the previous developments, we build a certified Quantifier elimination procedure for *real closed field*. *Quantifier elimination* is a standard way of proving the decidability of first order theories. The original result is from Tarski (cf [10])

We base the development of this procedure and its certificate on a paper proof that can be found at the beginning of [1]. The method consists in eliminating one existential variable at a time. We first consider the case of one variable,

and generalize to the case of multiple variable. However, this generalization poses problems in encoding the good structures to make the induction possible. But we have great hopes that the method we used for *quantifier elimination* on *algebraically closed fields* will work (cf [3]).

This proof will ensure the decidability of the first order theory of *real closed field*. However, this procedure is not efficient enough to compute the solution of basic concrete problems. But we hope the development of this library will ease the proof of more subtle quantifier elimination. One example of such procedures is the *cylindrical algebraic decomposition*, already implemented in Coq, but not yet certified (cf [9]).

6 Conclusion

We showed how we took advantage of boolean reflexion to ease reasoning over these structure where at least equality and ordering are decidable. We also showed that the way we state the lemmas is very important from a user point of view. We saw that we should state as much equalities as possible. That we may sometimes introduce some inductive type to write a specification and use it to make case analysis. But also that naming conventions are important for the user to find its path. We also propose some ways to state concepts like neighborhood and interval in a way that eases its use as much as possible.

The major drawback of this work is the lack of automation. But we hope that the constructions it still helps to make leads to the formalization of a reflexive procedure for deciding formula with comparison (like the *cylindrical algebraic decomposition*). This seems to be a natural and very powerful enhancing of this work. Nevertheless, before building such a tactic, we can hope for some incomplete automation that would be able to solve simpler facts. An example of such a tactic already implemented in Coq is the `fourier` tactic, but it is not yet compatible with our development for technical reasons.

A good thing to make this work a little more concrete would be to instantiate the structure of real closed field we defined. This can be done by building the real closure of rational numbers, for example.

Another natural extension of this work would be to generalize the analytic part of the formalization. This means we could try to do a little more than analysis for polynomials. We showed how some standard notions of real analysis can be represented in Coq, but when its formalization is constructive. To achieve that, we exploited the very strong properties of polynomials, but it may be possible to abstract a bit more from this properties. For example, we could use polynomial approximations to extend our results. We could head to uniform polynomial approximation or to approximation using series (cf [2])

Acknowledgments

The author wish to thank Assia Mahboubi for her regular feedback and contributions, both for the *order library* and its applications to *real closed fields*. But also for numerous comments and suggestions she made about this paper. I also which to thank George Gonthier for the discussions we had that helped the development, thanks to his advises and knowledge. Finally, I which to thank

François Bobot for his last minute relecture and suggestions.

References

[1] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal Proof of a Wave Equation Resolution Scheme: the Method Error. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the first Interactive Theorem Proving Conference (ITP)*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, July 2010. Springer. (merge of TPHOL and ACL2).

[3] Cyril Cohen and Assia Mahboubi. A formal quantifier elimination for algebraically closed fields. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick Ion, Laurence Rideau, Renaud Rioboo, and Alan Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin / Heidelberg, 2010.

[4] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[5] Herman Geuvers and Milad Niqui. Constructive reals in coq: Axioms and categoricity. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 725–726. Springer Berlin / Heidelberg, 2002.

[6] Georges Gonthier. A computer-checked proof of the four-colour theorem. Available at `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[7] Georges Gonthier. Notations of the four colour theorem proof. Available at `http://research.microsoft.com/~gonthier/4colnotations.pdf`.

[8] Cezary Kaliszyk and Russell O'Connor. Computing with classical real numbers. *CoRR*, abs/0809.1644, 2008.

[9] Assia Mahboubi. Implementing the cylindrical algebraic decomposition within the coq system. *Mathematical. Structures in Comp. Sci.*, 17(1):99–127, 2007.

[10] 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.