Formalization in Coq of polynomial interpretations on rationals

Kim Quyen LY

Supervisor: Frédéric BLANQUI

October 2, 2015

Contents

1 Introduction

This is the report of my internship in the project FORMES[1]. This internship took place at TsingHua University[2](Beijing, China) within the INRIA project-team FORMES which is part of the LIAMA[3] Consortium, the Sino-French Laboratory for Computer Science, Automation and Applied Mathematics. This internship took 5 month (May - Oct, 2010). In this report, I will talk about *polynomial interpretations, termination, `Coq` and how to formalize in `Coq` polynomial interpretations on rational numbers.*

Termination is an important property required for total correctness of programs and algorithms. We want to evaluation of a given program definitely terminate or not or simply find the answer "*program does terminate or not*" whenever this is possible. The answer is "*may be*" or continue working infinitely long. Because of that many criteria and tools have been developed over the last years. Their is also an annual international termination competition[4]. For these tools to be used in critical systems and proof assistants, their results must be certified. These are the reason the project CoLoR was born, this project aims at certifiying the termination proofs found by these tools. I will talk more about it in section 5. CoLoR is a Coq library on rewriting and termination. `Coq`[5], developed at INRIA (France), is an interactive theorem prover. It allows the expression of mathematical assertions, mechanically checks proofs of these assertions, helps to find formal proofs, and extracts a certified program from the constructive proof of its formal specification. You will find more information about `Coq` in section 4.

My main subject was working on polynomials in CoLoR. For instance, a termination tool could say that a rewrite system is terminating by providing a polynomial interpretation. Then, we would try to automatically generate in `Coq` a proof that this interpretation statisfies the conditions required by the theorem on polynomial interpretations. Currently, the CoLoR library only provides polynomials on the set $\mathbb{Z}$ of integers.

My work was to generalize the current development of CoLoR to any (ordered) ring structure on a setoid, that is, a set equipped with an arbitrary decidable equivalence relation; prove the wellfoundedness of the δ-ordering on the set $\mathbb{Q}^+$ of non-negative rationals; and improve the criterion to check the monotony of polynomials. My work is detailed in section 6.

Finally, I would like to give my special thanks to my supervisor Prof. Frédéric BLANQUI, Sidi Ould Biha and all members in FORMES project.

[1]http://formes.asia
[2]http://www.thss.tsinghua.edu.cn/index_en.asp
[3]http://liama.ia.ac.cn
[4]http://www.termination-portal.org
[5]http://coq.inria.fr

2 Term rewriting systems

In this section, we introduce the notions of term rewriting. For more details we refer to for instance [1, 9]. Here we restrict to first-order rewriting only. A *term rewriting system* (TRS) consists of rewrite rules formed with terms. For that we need to define terms, which correspond to terms of first-order predicate logic. They are defined over a first-order signature.

A relation $>$ is a *strict partial order* if it is irreflexive and transitive. We say that the relation $\geqslant$ is a *quasi-order* if it is reflexive and transitive.

A *signature* F is a non-empty set of *function symbols*, equipped with an *arity function*[6], arity : $F \to \mathbb{N}$, indicating how many arguments each function symbol expects.

The set of *terms* over a signature F and over an infinitive set of variables V, disjoint from F, is denoted by $T(F, V)$ and defined inductively as:

- $x \in T(F, V)$, for $x \in V$
- $f(t_1, ..., t_n) \in T(F, V)$, for $f \in F$, arity $(f) = n$ and $t_1, ...t_n \in T(F, V)$.

A *rewrite rule* is a pair of terms (l, r) with $l, r \in T(F, V)$. A rewrite rule (l, r) is usually written as $l \to r$.

The left-hand side is not a variable and every variable occurring in the right-hand side must also occur in the left-hand side of a rule. Term rewriting system is simply a set of rules.

A *term rewriting system* is a pair (F, R) of a signature F and a set of rewrite rules R. The signature is usually left implicit and a term rewriting system is identified with its set of rewrite rules.

A *substitution* is a mapping from variables to terms σ: $V \to T(F, V)$. An application of a substitution σ to a term t is written as $t\sigma$ and defined as follows:

$$x\sigma = \sigma(x)$$

$$f(t_1, ..., t_n)\sigma = f(t_1\sigma, ..., t_n\sigma)$$

A relation $\to$ is *closed by substitution* if, for all terms t, u and substitution σ, $t\sigma \to u\sigma$ whenever $t \to u$.

A *context* is a term with a unique occurrence of a distinguished variable $[]$. A relation $\to$ is *closed by context* if, for all terms t, u and context C, $C[t] \to C[u]$ whenever $t \to u$, $C[t]$ being C with $[]$ replaced by t.

The *rewrite relation* $\to_R$ is the smallest relation closed by substitution and context containing R. If $t \to_R u$ then there is a rule $l \to r \in R$, a substitution σ and a context C such that $t = C[l\sigma]$ and $u = C[r\sigma]$. We then say that there is a rewrite step from t to u or, simply, that t rewrites to u.

Note that we will often write $\to$ instead of $\to_R$ if the TRS R is clear from the context.

[6]An arity function (or operation) is the number of arguments or operands that the function takes.

3 Termination using polynomial interpretations

Termination is an important concept in term rewriting.

A TRS R is terminating if the relation $\rightarrow_R$ is terminating, and a binary relation $\rightarrow$ is *terminating* or *strongly normalizing*, SN($\rightarrow$), if it is well-founded, *i.e.,* if there exists no infinite sequence $t_0, t_1, ...$ such that $t_i \rightarrow t_{i+1}$ for all $i \in \mathbb{N}$.

Definition 1 A TRS R is called *terminating* if there is no infinite reduction $t_1 \rightarrow_R t_2 \rightarrow_R ..., i.e.,$ SN $(\rightarrow_R)$.

3.1 Reduction orderings

We introduce the notion of reduction orderings and a theorem employing them for proving termination.

Proving termination of a TRS is equivalent to finding a well-founded relation $>$ on terms that is stable by substitution and context (i.e., a *reduction ordering*) and *compatible* with the rules of the TRS, i.e., such that $l > r$ for all rules $l \rightarrow r$ of the TRS.

Definition 2 A *reduction ordering* is well-founded relation closed by substitution and context.

Theorem 3 A TRS R *is terminating*, SN(R), *iff there exists a reduction ordering* $>$ *such that* $R \subseteq >$.

Proof.

First we prove: $SN(R), \exists > \Rightarrow R \subseteq >$.
A TRS R is terminating if the relations $\rightarrow_R$ is terminating, i.e. $SN(R)$ if $SN(\rightarrow_R)$, i.e. there is no infinite sequence (t_n) such that: $t_n \rightarrow_R t_{n+1}, \forall n \in \mathbb{N}$. So that, $R \subseteq \rightarrow_R$. We assume that $> = \rightarrow_R$, so we have $R \subseteq >$.

Second we prove: $\exists >, R \subseteq > \Rightarrow SN(R)$.
Let D be an arbitrary non-empty domain. Equivalently, We can prove $\neg SN(R) \Rightarrow \forall >, R \nsubseteq >$. Let (t_n) be an infinite sequence of $\rightarrow_R$ steps, such that: $t_n \rightarrow_R t_{n+1}, \forall n \in \mathbb{N}$. Since, $R \subseteq >$ and $>$ is close by substitution and context. Then we have $\rightarrow_R \subseteq >$, so (t_n) is an infinite sequence of $>$ steps, too.

■

3.2 Interpretations

We now come to the interpretation of terms into some non-empty domain D given an interpretation function: $I_f : D^n \rightarrow D$ for each function symbol f of arity n. This gives D a structure of a F-algebra where F is the signature.

Given a valuation ρ: $\chi \rightarrow D$ for the variables of t, the interpretation of a term t written $[\![t]\!]_\rho$, is the recursive application of the interpretation functions I_f:

- $[\![x]\!]_\rho = \rho(x)$
- $[\![f(t_1,\ldots,t_n)]\!]_\rho = I_f([\![t_1]\!]_\rho,\ldots,[\![t_n]\!]_\rho)$

Definition 4 Given an interpretation I and a relation $>_D$ on the domain D of I, the relation on terms associated to I is $t >_I u$ if $[\![t]\!]_\rho >_D [\![u]\!]_\rho$ for all valuation $\rho : \chi \to D$.

Note that $>_I$ is well-founded when $>_D$ is well-founded, it is always stable by substitution and it is stable by context if the functions I_f are monotone wrt $>_D$ in every argument.

Lemma 5 $\forall t, \sigma, \mu, [\![t\sigma]\!]\mu = [\![t]\!]\sigma\mu$.

Proof.
We proof by structural induction on t (or well-founded induction on the size of t).

- $t = x$
  $[\![x\sigma]\!]\mu = [\![x]\!]\sigma\mu$ (by definition 4)
- $t = f(t_1,\ldots,t_n)$. $[\![t\sigma]\!]\mu = I_f([\![t_1\sigma]\!]\mu,\ldots,[\![t_n\sigma]\!]\mu)$.
  By induction hypothesis, for all i, $[\![t_i\sigma]\!]\mu = [\![t_i]\!]\sigma\mu$.

■

Theorem 6 The relation $>_I$ is a reduction ordering if $D \neq \emptyset$, $>_D$ is well-founded and every $I_f : D^n \to D$ is monotonic in each variable.

Proof.
First we prove: $>_I$ is closed by substitution.
$>_I$ is closed by substitution if, for all $t, u, \sigma, t >_I u \Rightarrow t\sigma >_I u\sigma$ (by definition 4).
Let t, u, σ such that $t >_I u$.
We have $t\sigma >_I u\sigma$ if for all valuation μ, $[\![t\sigma]\!]\mu >_D [\![u\sigma]\!]\mu$ (by definition 4).
From the lemma 5, we have:
$[\![t\sigma]\!]\mu = I_f([\![t_1]\!]\sigma\mu,\ldots,[\![t_n]\!]\sigma\mu) = [\![t]\!]\sigma\mu$.

Second we prove: $>_I$ is closed by context if every $I_f : D^n \to D$ is monotonic in each variable.
Let $f, t_1, \ldots, t_n, t'_k$ such that $t_k >_I t'_k$.
We have to prove that $ft_1 \ldots t_{k-1}t_kt_{k+1} \ldots t_n >_I ft_1 \ldots t_{k-1}t'_kt_{k+1} \cdots t_n$.

Let μ be a valuation. We have:

$[\![ft_1 \ldots t_{k-1}t_kt_{k+1} \ldots t_n]\!]\mu$
$= I_f([\![t_1]\!]\mu,\ldots,[\![t_{k-1}]\!]\mu,[\![t_k]\!]\mu,[\![t_{k+1}]\!]\mu,\ldots,[\![t_n]\!]\mu)$
$>_D I_f([\![t_1]\!]\mu,\ldots,[\![t_{k-1}]\!]\mu,[\![t'_k]\!]\mu,[\![t_{k+1}]\!]\mu,\ldots,[\![t_n]\!]\mu)$

$= [\![f t_1 \dots t_{k-1} t'_k t_{k+1} \dots t_n]\!]\mu$

Because I_f is monotonic in its k-th argument and $[\![t_k]\!]\mu >_D [\![t'_k]\!]\mu$ by assumption.

Third we prove: $>_I$ is well-founded if $>_D$ is well-founded, i.e. $SN(>_D) \Rightarrow SN(>_I)$, and $D \neq \emptyset$.

Equivalently, we can prove $\neg SN(>_I) \Rightarrow \neg SN(>_D)$. So, let (t_n) be an infinite sequence of $>_I$ steps, i.e. for all n and valuation ρ, $[\![t_n]\!]\rho >_D [\![t_{n+1}]\!]\rho$. Since $D \neq \emptyset$, there is $d \in D$. Then, let $\rho(x) = d$ the constant valuation equal to d. For all n, $[\![t_n]\!]\rho >_D [\![t_{n+1}]\!]\rho$ and $([\![t_n]\!]\rho)$ is an infinite sequence of $>_D$ steps.

■

3.3 Polynomial interpretations on $\mathbb{Z}$

Polynomial interpretations, *i.e* when the functions I_f are polynomials, are a useful technique for proving termination of term rewrite systems. In an automated setting, termination tools are concerned with polynominals whose coefficients are initially unknown and have to be instantiated suitably such that the resulting concrete polynominals statisfy certain conditions.

Theorem 7 $>_I$ is a reduction ordering if I is a polynomial interpretation on Z such that:

- for all f, all the coefficients of I_f are non-negative
- for all f of arity n and variable x_i $(i \leq n)$, the coefficient of x_i is positive
- $>_D$ is well-founded on non-negative elements

Proof.
First note that I is well-defined since $I_f = eval(P_f) : D^n \to D$ becaues, for all f, all the coefficients are non-negative.
After previous theorem, we have to check:

- $D \neq \emptyset$. Because $D = \mathbb{Z}^+ = \mathbb{N}$.
- $>_D$ is well-founded by assumption.
- I_f are monotonic in every argument.

  Let $x_1, \dots, x_n, x'_k$ such that $x_k >_D x'_k$.

  We have to prove that $I_f(x_1, \dots, x_n) >_D I_f(x_1, \dots, x'_k, \dots, x_n)$, i.e. $\delta = I_f(x_1, \dots, x_n) - I_f(x_1, \dots, x'_k, \dots, x_n) >_D 0$.

  Since $x_k >_D x'_k$, there is $\delta > 0$ such that $x_k = x'_k + \delta$. In addition, there are two polynomials R and S such that $I_f = x_k R + S$ and x_k does not occur in S (Euclidian division on $R[X]$). So, $I_f(\dots, x_k, \dots) =$

$I_f(\ldots, x'_k + \delta, \ldots) = (x'_k + \delta)R(\ldots, x'_k + \delta, \ldots) + S$ and $\delta = x'_k(R(\ldots, x'_k + \delta, \ldots) - R(\ldots, x'_k, \ldots)) + \delta R(\ldots, x'_k + \delta, \ldots)$.

All the coefficients of $R(\ldots, x'_k + \delta, \ldots) - R(\ldots, x'_k, \ldots)$ are non-negative since all the coefficients of R are non-negative (proof not given). So, $\delta \geq 0$. Moreover, $\delta = 0$ only if $R = 0$ which is not possible by assumption.

■

In addition, $l >_I r$ if all the coefficients of $P_l - P_r - 1$ are non-negative.

4 Presentation of Coq

In this section, we briefly present Coq. The Coq system is a computer tool for verifying theorem proofs.

Coq is not an automated theorem prover but includes automatic theorem proving tactics and various decision procedures. Coq implements a functional programming language.

Coq is also used to develop libraries of advanced mathematical theorems in both constructive and classical form.

Working in Coq you can discovery of a difficult but exciting world when you can enjoy of the last QED.

To discovery more about Coq you can find in its web-page or read the book "The Coq'Art" by Yves Bertot and Pierre Castéran [2].

4.1 A brief overview

The Coq system provides a language in which one handles formulas, verify that they are well-formed, and prove them. The first thing you need to know about Coq is how you can check whether a formula is well-formed. The command Check is use for this purpose.

```
Check True.
True : Prop
```

Now we give an example to describe how to prove something in Coq. There are two approachs to construct a proof, first new theorems can be constructed by combining existing theorems and using the Definition keyword to associate these expressions to constants. The other approach is known as *goal directed proof*: This first approach is seldom used.

1. the user enters a statement that he wants to prove, using the command Theorem or Lemma,

2. the Coq system displays the formula as a formula to be proved, possibly giving a context of local facts that can be used for this proof,

3. the user enters a command to decompose the goal into simpler ones,

4. the Coq system displays a list of formulas that still need to be proved,
5. back to step 3.

When there are no more goals the proof is complete, it needs to be saved, this performed when the user sends the command Qed. The commands that are especially designed to decompose goals into list of simpler goals are called *tactics*.

Here is an example:

```
Theorem example : forall a b : Prop, a /\ b -> b /\ a.
Proof.
intros a b H. tauto.
Qed.
```

This theorem can be proved in many ways, but a quick one would simply uses the tauto tactic directly. There is an important collection of tactics in the Coq system, each of which is adapted to a shape of goal. It is worthwhile remembering a collection of tactics for the basic logical connectives.

Some automatic tactics are also provided for a variety of purposes, tauto is often useful to prove facts that are tautogogies in propositional intuitionistic logic (try it whenever the proof only involves manipulations, conjunction, disjunction, and negation); auto is an extensible tactic that tries to apply a collection of theorems that were provided beforehand by the user, eauto is like auto, it is more powerful but also more time-consuming, ring and ring_nat mostly do proofs of equality for expressions containing addition and multiplication, omega proves formulas in Presburger arithmetic.

4.2 Inductive types

Inductive types could also be called algebraic types or initial algebras. They are defined by providing the type name, its type, and a collection of constructors. Inductive types can be parameterized and dependent. We will mostly use parameterization to represent polymorphism and dependence to represent logical properties.

Defining inductive types

Here is an example of an inductive type definition:

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

The type *nat* is defined as the least *Set* containing O and closed by the *S* constructor. The Coq system automatically associates an inductive principle to this inductive type named *name*_ind. For natural number, this is nat_ind.

Pattern matching

Pattern matching makes it possible to describe functions that perform a case analysis on the value of an expression whose type is an inductive type.

```
Definition andb (b1 b2 : bool) : bool :=
  match b1 with
    | true => b2
    | false => false
  end.
```

For this example, we write a function that return the boolean value false when its argument is false and return b2 when its is true.

Inductive properties

Inductive types can be dependent and when they are, they can be used to express logical properties. When defining inductive type like nat, Z we declare that this constant is a type. When defining a dependent type, we actually introduce a new constant which is declared to be a function from some input type to the type of types.

Like other inductive types, inductive properties are equipped with an inductive priciple, which we can use to perform proofs. Inductive properties can be used to express very complex notions.

4.3 Proofs

In this section we will present some approachs use to do proof in Coq.

Proof by simplification

The proofs of these claims: use the function's definition to simplify the expressions on both sides of the = and notice that they become identical. The same sort of "proof by simplification" can be used to prove more interesting properties.

Here is an example using proof by simplification:

```
Theorem plus_0_n : forall n : nat, plus 0 n = n.
Proof.
simpl. reflexivity.
Qed.
```

Proof by rewriting

Here is an example we proof by rewriting.

```
Theorem plus_example : forall n m : nat,
  n = m -> plus n n = plus m m.
Proof.
```

```
intros n m H.
rewrite H.
reflexivity.
Qed.
```

The first line of the proof moves variables n, m and the hypothesis $n = m$ into the context and gives it the name H. The second tells Coq to rewrite the current goal (plus n n = plus m m) by replacing the left side of the equality hypothesis H with the right side. There are two ways to rewrite, first we would like to rewrite from left to right, you can write rewrite H or rewrite → H, and another is rewrite from right to left, you can use rewrite ← H.

Proof by induction

The most general kind of proof that one can perform on inductive types is proof by induction.

When we prove a property of the elements of an inductive type using a proof by induction, we actually consider a case for each constructor. There is a twist: when we consider a constructor that has arguments of the inductive type, we can assume that the property we want to establish holds for each of these arguments.

```
Theorem plus_assoc: forall x y z : nat,
  (x + y) + z = x + (y + z).
Proof.
intros x y z.
elim x. rewrite plus_0_n; trivial.
intros x' H. rewrite (plus_Sn_m x' y);
rewrite (plus_Sn_m (x' + y) z).
rewrite plus_Sn_m; rewrite H; trivial.
Qed.
```

To understand this proof first of all we need to know what is plus_0_n, plus_Sn_m. We can use the command Check.

```
Check plus_0_n.
plus_0_n : forall n : nat, 0 + n = n
Check plus_Sn_m.
plus_Sn_m : forall n m : nat, S n + m = S (n + m)
```

We can see that a proof by induction on x should work, when x is zero, two uses of plus_0_n should help to show that both members of the equality are the same. When x is "Sx" and the induction hypothesis holds for x', three uses of plus_Sn_m and the induction hypothesis should make the proof complete.

```
  x : nat
  y : nat
  z : nat
  ============================
   0 + y + z = 0 + (y + z)
```

The tactic elim is the tactic that is used to indicate that a proof by induction is being done.

This tactic generates two goals, a first goal we can rewriting with the theorem plus_0_n, and a second goal

```
  x : nat
  y : nat
  z : nat
  ============================
   forall n : nat, n + y + z = n + (y + z)
    -> S n + y + z = S n + (y + z)
```

To make it more readable, we introduce the variable x' and the hypothesis H.

```
  x : nat
  y : nat
  z : nat
  x': nat
  H : x' + y + z = x' + (y + z)
  ============================
   S x' + y + z = S x' + (y + z)
```

We use the theorem plus_Sn_m twice to make it obvious that the left-hand side of the equality is S applied to the left-hand side of the induction hypothesis.

```
  x : nat
  y : nat
  z : nat
  x': nat
  H : x' + y + z = x' + (y + z)
  ============================
   S (x' + y + z) = S x' + (y + z)
```

It is now obvious that rewriting again with plus_Sn_m and with the induction hypothesis makes it possible to colude.

```
rewrite plus_Sn_m; rewrite H; trivial.
Qed.
```

4.4 Numbers

In the Coq system, most usual datatypes are represented as inductive types and packages provide a variety of properties, functions, and theorems around these datatypes. The package named Arith contains a lot of theorems about natural numbers, the package named ZArith provides represent integers. The package named QArith provides descriptions of rational numbers. The support for on real numbers in the Coq system is also quite good, but real numbers are not (and cannot) be represented using an inductive type. The package to load is Reals.

4.5 Data structures

The two-element boolean type is an inductive type in the Coq system, true and false are its constructors. Most ways to structure data together are also provided using inductive data structures, a commonly used datatype is the type of lists. This type is polymorphic, in the sense that the same inductive type can be used for lists of natural numbers, lists of boolean values, or lists of other lists. This type is not provided by default in the Coq system, it is necessary to load the package List using the Require command to have access to it. Here is an example a list of natural number.

```
Require Import List.
Check (cons 3 (cons 2 (cons 1 nil))).
3 :: 2 :: 1 :: nil : list nat
```

This example also shows that the notation :: is used to rerpresent the cons function in an infix fashion. The List package also provides a list concatenation function named app, with ++ as infix notation, and a few theorems about this function.

5 Current formalization in CoLoR

In this section we will present the CoLoR project and the formalization of the polynominal interpretation based termination methods developed within the CoLoR project.

The goal of the project CoLoR is to verify the results produced by termination provers with the use of the Coq proof assistant, we have presented in section 4. This is achieved by means of certificates, that is a transcription of termination proofs in a dedicated formate.

CoLoR consists of three parts:

1. TCG (Termination Certificate Grammar): a formal grammar for termination certificates.

2. CoLoR (Coq library on Rewriting and Termination): a library of results on termination of rewriting, formalized in Coq.

3. Rainbow: a tool for transforming termination certificates in the TCG format into Coq scripts certifying termination by employing results from CoLoR.

   For more information about the project we assume to go its web-page:

   http : //color.inria.fr

Rainbow supports various advanced termination criteria used in mordern automated termination provers. There are two distinct approaches to do in this: one is using a shallow embedding [3] and another is a deep embedding. CoLoR uses deep embeddings only.

5.1 Terms and rewriting

The CoLoR library contains many functions and theorems on basic data structure like lists, vectors, polynomials, matrices and finite multisets.

CoLoR provides various notions of terms: strings, first-order terms with symbols of fixed arity,... Algebraic terms are inductively defined from a signature (module ASignature) defining the set of symbols, the arity of each symbol, a boolean function saying if two symbols are equal or not.

```
Notation variable := nat (only parsing).
Record Signature : Type := mkSignature {
   symbol :> Type;
   arity : symbol -> nat;
   beq_symb : symbol -> symbol -> bool;
   beq_symb_ok : forall x y, beq_symb x y = true <-> x = y}
```

The type of algebraic terms on a signature (module ATerm) is defined as follows:

```
Inductive term : Type :=
  | Var : variable -> term
  | Fun : forall f : Sig, vector term (arity f) -> term.
```

The type of vectors (also called arrays or dependent lists) with elements of type A:

```
Inductive vector : nat -> Type :=
  | Vnil : vector 0
  | Vcons : forall (a:A)(n:nat), vector n -> vector (S n).
```

Rewrite relations are defined from sets of rules, a rule simply being a pair of terms:

```
Record rule : Type := mkRule {lhs : term; rhs : term}.
```

Contexts are defined as terms with a unique hole in a similar way.

```
Notation terms := (vector term).
Inductive context : Type :=
  | Hole : context
  | Cont : forall (f : Sig)(i j : nat), i + S j = arity f ->
    terms i -> context -> terms j -> context.
```

The standard rewrite relation generated from a list R of rewrite rules is then defined as follows:

```
Definition red u v := exists l r c s, In (mkRule l r) R /\
   u = fill c (sub s l) /\ v = fill c (sub s r).
```

We have red : term → term → Prop, so red is a relation on terms. We have a reduction step red u v, written as $u \to_R v$, whenever there exist a term l, a term r, a context C and a subtitution σ such that $l \to r \in R$, $u = C[l\sigma]$ and $v = C[r\sigma]$.

5.2 Interpretations

The interpretation of terms into some non-empty domain D:

```
Definition naryFunction A B n := vector A n -> B.
Definition naryFunction1 A := @naryFunction A A.
Record interpretation : Type := mkInterpretation {
  domain :> Type;
  some_elt : domain;
  fint : forall f : Sig, naryFunction1 domain (arity f)}.
```

The substitution is then nothing but an interpretation on the domain of terms by taking $I_f(t_1, ..., t_n) = f(t_1, ..., t_n)$.

```
Definition I0 := mkInterpretation (Var 0) (@Fun Sig).
Definition substitution : valuation I0.
Definition sub : substitution -> term -> term := @term_int Sig I0.
```

The formalization used a class of interpretations on well-founded domain of natural numbers: polynomial interpretations. Currently, formalization of polynomial in CoLoR is integer polynomials and it is simple.

5.3 Polynomials

Polynomial interpretations were contributed to the CoLoR library by Sébastien Hinderer [4]. This library on polynomials with multiple indeterminates and integer coefficients.

Example: $[]$, $[0,[1]]$ and $[(1,[1]),(-1,[1])]$ all represent $0 = 0x = 1x - 1x$.

Polynomial represented by using monomial because of the simple structure of the monomial basic. $p = \sum_{n}^{i=0} a_i x^i$.

The type of polynomials depends on the maximum number n of variables. A polynomial is represented by a list of pairs made of an integer and a monomial. A monomial begin a vector of size n made of the power of each variable.

```
Notation monom := (vector nat).
Definition poly n := list (Z * monom n).
```

Example: $3XY + 2X^2 + 1$ is represented by $[(3, [|1; 1|]); (2, [|2; 0|]); (1, [|0; 0|])]$

The coefficient of a monomial:

```
Fixpoint coef n (m : monom n) (p : poly n) {struct p} : A :=
  match p with
    | nil => 0
    | cons (c,m') p' =>
      match monom_eq_dec m m' with
        | left _ => c + coef m p'
        | right _ => coef m p'
      end
  end.
```

The module Polynom provides basic operations on polynomials: addition, subtraction, multiplication, power, composition, evaluation to an integer ($\mathbb{Z}$) given values for variables and theorems on monotony.

A polynomial interpretation consits in associating to every function symbol of arity n, an integer polynomial with n variables (module APolyInt).

```
Definition PolyIterpretation := forall f : Sig, poly (arity f).
```

Module PolyInt contains definition and properties about polynomial interpretations for algebraic terms. A polynomial interpretation given by, for each symbol of arity n, a monotonic polynomial on n indeterminates (the monotonicity of a polynomial P of n indeterminates is ensured by requiring that, for all $i \leq n$, the coefficient of x_i in P is positive). A term with n variables is then interpreted by a polynomial $\|t\|$ with n indeterminates. By using the evaluation function of polynomials in the domain D of non-negative integers, we get an interpretation in the usual sense, where a symbol of arity n is interpretated by a function from D^n to D. It is also proven that the ordering on terms obtained by comparing the interpretations ($t > u$ if $\|t\| > \|u\|$) is a reduction ordering. By the Manna-Ness theorem, if R is included in $>$ then $red(R)$ is terminating. And, for proving that $l > r$, it suffices to check that the coefficients of $\|l\| - \|r\| - 1$ are non-negative.

CoLoR also provides a simple test for (strict) monotonicity, by testing that each monomial x_i is (strictly) positive.

In contrast to matrices or multisets, polynomials are not yet defined as a functor building a structure for polynomials given a structure for the coeffcients. And my work is change this in order to be able to certify proofs using polynomial interpretations with rational or real coeffcients.

6 My contributions

6.1 A generic interface for rings

Our first goal is to be able to use polynomials on any ring structure. A *ring* consists of a carrier D, two designated elements d_0, $d_1 \in D$ and two binary operations $\oplus$, $\otimes$ on D, such that both $(D, d_0, \oplus)$ and $(D, d_1, \otimes)$ are commutative monoids and multiplication distributed over addition: $\forall x, y, z \in D : x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$.

Coq has built in notion of ring defined as a structure and a tactic ring for proving by reflection polynomial equations.

But Coq has aslo a module system, this module system uses structures, signatures, and parametric modules (also called *functors*), that is already used in CoLoR but encloses the semi-ring specification within a module providing a real encapsulation and modularization. It aslo allows to prove a number of results following from the specification of a semi-ring that will automatically be available for any instantiation to an actual semi-ring. So we defined a module type for rings extending the Coq built in ring structure. An alternative would be to use type classes.

To start defining a signature, we use the keywords "Module Type" followed by the name of the signature. Types and operations are declared with the Parameter command, propositions are declared with the Axiom command; a signature description is ended by the keyword End with the signature name, as for sections.

To define a ring structure on A, we must provide an addition, a multitplication, an opposite function and two unities 0 and 1. We must then prove all theorems that make (A0, A1, Aadd, Amul, Aopp, eqA) a ring structure, and pack them with the ring_theory constructor.

To illustrate this, we give the signature for Ring.

- a type A,
- a constants A0, A1 is for 0 and 1,
- a function Aadd : $A \to A \to A$ is for addition,
- a function Amul : $A \to A \to A$ is for multiplication,
- a function Aopp : $A \to A$ is for opposite,
- a function Asub : $A \to A \to A$:= fun x y $\Rightarrow$ Aadd x (Aopp y) is a definition of substration,
- a function Aring : ring_theory A0 A1 Aadd Amul Aopp eqA is for the ring theory, ie. all the properties required by Aadd, Amul, Aopp, Asub for A to be a ring.

A signature can also contain a reference to another module with another signature. For instance, we want to consider rings for setoids equality eqA of type A which means that a set equipped with an arbitrary decidable equivalence (with only equality).

We define a module types for setoids with deciable equality. Setoid_Theory A eqA, for the declaration of setoids where eqA is a congruence relation, A is a type, such that:

```
Definition Setoid_Theory := @Equivalence.

Module Type SetA.
Parameter A : Type.
End SetA.

Module Type Eqset.
Parameter A : Type.
Parameter eqA : A -> A -> Prop.
Notation "X =A= Y" := (eqA X Y) (at level 70).
Parameter sid_theoryA : Setoid_Theory A eqA.
...
End Eqset.
```

Eqset_dec is a module type of decidable equality for type A

```
Module Type Eqset_dec.
Declare Module Export Eq: Eqset.
Parameter eqA_dec : forall x y, {x =A= y} + {~x =A= y}.
End Eqset_dec.
```

One function can respect several different relations and thus it can be decleared as a morphism having multiple signatures. To declare multiple signatures for a morphism, repeat the Add Morphism command, following signatures: eqA ==> eqA ==> eqA (eqA being the equality, the special arrow ==> is used in signatures for morphisms that are both convariant and contravariant).

```
Module Type RingType.
  Declare Module Export ES : Eqset_dec.
  Parameter A0 : A.
  Parameter A1 : A.
  Parameter Aadd : A -> A -> A.
  Add Morphism Aadd with signature eqA ==> eqA ==> eqA as Aadd_mor.
  Parameter Amul : A -> A -> A.
  Add Morphism Amul with signature eqA ==> eqA ==> eqA as Amul_mor.
  Parameter Aopp : A -> A.
  Add Morphism Aopp with signature eqA ==> eqA as Aopp_mor.
  Definition Asub : A -> A -> A := fun x y => Aadd x (Aopp y).
  Add Morphism Asub with signature eqA ==> eqA ==> eqA as Asub_mor.
  Parameter Aring : ring_theory A0 A1 Aadd Amul Asub Aopp eqA.
End RingType.
```

Next we would like to build a module for ring theory, integer numbers and rational numbers.

We can use the keywords "Module RingTheory", to opens the description of a module with the name RingTheory without specifying a signature. We give some notations by using the command Notation which these notations we can easily write the program, to declare the setoids we use the keywords "Add Setoid", after we adapt all of it to a ring by using a command "Add Ring". Due to backward compatibility reasons, the following syntax for the declaration of setoids and morphisms is also accepted (Add Setoid A eqA sid_thoeryA as A_Setoid), where eqA is a congruence relation without parameters, A is its carrier and sid_theoryA is an object of type.)

```
Module RingTheory (Export R : RingType).
  Notation "0" := A0.
  Notation "1" := A1.
  Notation "x + y" := (Aadd x y).
  Notation "x * y" := (Amul x y).
  Notation "- x" := (Aopp x).
  Notation "x - y" := (Asub x y).
  Add Setoid A eqA sid_theoryA as A_Setoid.
  Add Ring Aring : Aring.
```

We can define some fields are useful by using the regular commands Definition, Fixpoint, Theorem, Lemma and so on. The process ends with a closing command "End RingTheory".

```
 Fixpoint power x n {struct n} :=
    match n with
      | 0 => 1
      | S n' => x * power x n'
    end.
 Lemma power_add : forall x n1 n2, x ^ (n1 + n2) =A= x ^ n1 * x ^ n2.
 Lemma Aadd_0_r : forall x, x + 0 =A= x.
 Lemma Amul_0_l : forall x, 0 * x =A= 0.
End RingTheory.
```

Rings over integer numbers $\mathbb{Z}$

We continue define a module for integer numbers by using the keywords "Module Int <: SetA", it means that open the description of a module with the name Int as a compatible module with the signature SetA. We want to use the library on Integers, we need to load the package ZArith.

```
Require Import ZArith.
Module Int <: SetA.
  Definition A := Z.
End Int.
```

To obtain an implementation with Eqset_def, we can simply apply the functor Eqset_def to the module Int. We now choose to only verify that an implementation with integers only satifies the specification Eqset_dec without masking definitions. We simply use the operator <:

```
Module IntEqset := Eqset_def Int.
Module IntEqset_dec <: Eqset_dec.
  Module Export Eq := IntEqset.
  Definition eqA_dec := dec_beq beq_Z_ok.
End IntEqset_dec.
```

We now implement with integers only satifies in RingType, and mapping all the parameters of type A into integers (or type Z).

```
Module IntRing <: RingType.
Module Export ES := IntEqset_dec.
  Add Setoid A eqA sid_theoryA as A_Setoid.
  Definition A0 := 0%Z.
  Definition A1 := 1%Z.
  Definition Aadd := Zplus.
  Add Morphism Aadd with signature eqA ==> eqA ==> eqA as Aadd_mor.
  Definition Amul := Zmult.
  Add Morphism Amul with signature eqA ==> eqA ==> eqA as Amul_mor.
  Definition Aopp := Zopp.
  Add Morphism Aopp with signature eqA ==> eqA as Aopp_mor.
  Definition Asub : Z -> Z -> Z := fun x y => Zplus x (Zopp y).
  Add Morphism Asub with signature eqA ==> eqA ==> eqA as Asub_mor.
  Lemma Aring : ring_theory A0 A1 Aadd Amul Asub Aopp eqA.
End IntRing.
Module IntRingTheory := RingTheory IntRing.
```

Rings over rational numbers $\mathbb{Q}$

```
Require Import QArith.
Module Rat_Eqset <: Eqset.
  Definition A := Q.
  Definition eqA := Qeq.
  Definition sid_theoryA: Setoid_Theory A eqA.
End Rat_Eqset.
Module Rat_Eqset_dec <: Eqset_dec.
  Module Export Eq := Rat_Eqset.
  Lemma eqA_dec : forall x y, {eqA x y} + {~eqA x y}.
End Rat_Eqset_dec.
Module RatRing <: RingType.
  Module Export ES := Rat_Eqset_dec.
  Add Setoid A eqA sid_theoryA as A_Setoid.
  Definition A0 := 0#1.
  Definition A1 := 1#1.
  Definition Aadd := Qplus.
  Add Morphism Aadd with signature eqA ==> eqA ==> eqA as Aadd_mor.
  Definition Amul := Qmult.
  Add Morphism Amul with signature eqA ==> eqA ==> eqA as Amul_mor.
  Definition Aopp := Qopp.
  Add Morphism Aopp with signature eqA ==> eqA as Aopp_mor.
  Definition Asub: Q -> Q -> Q := fun x y => Qplus x (Qopp y).
  Add Morphism Asub with signature eqA ==> eqA ==> eqA as Asub_mor.
  Definition Aring := Qsrt.
End RatRing.
Module RatRingTheory := RingTheory RatRing.
```

6.2 A generic interface for ordered rings

Another goal we build the ring structure is to be able to use the ordered rings, which means that we can check some conditions for instance $1 > 0$ and so on.

We implement with order ring, with this order we able to certify proofs using polynomial intergretations with rational or real coefficients. We use a signature OrdRingType for types with a total decidable order. We also give a minimum axioms in this module, this means that we have a small set of functions and their specifications and those results are independent of a particular representation of multisets.

There are some components in this signature:

- a binary relation gtA ($>$) on A,
- declare a relation for morphism is iff
- a transitive of this relation gtA_trans,
- a irreflexive gtA_irrefl,
- a boolean function of this relation bgtA,
- a function checking the correctness of this condition bgtA_ok,
- a condition $1 > 0$, one_gtA_zero,
- a specification with addition and multiplication add_gtA_mono_r and mul_gtA_mono_r.

```
Module Type OrdRingType.
  Declare Module Export R : RingType.
  Module Export RT := RingTheory R.
  Parameter gtA : A -> A -> Prop.
  Notation "x >A y" := (gtA x y) (at level 70).
  Add Morphism gtA with signature eqA ==> eqA ==> iff as gtA_morph.
  Parameter gtA_trans : transitive gtA.
  Parameter gtA_irrefl : irreflexive gtA.
  Parameter bgtA : A -> A -> bool.
  Parameter bgtA_ok : forall x y, bgtA x y = true <-> x >A y.
  Parameter one_gtA_zero : 1 >A 0.
  Parameter add_gtA_mono_r: forall x y z, x >A y -> x + z >A y + z.
  Parameter mul_gtA_mono_r:
    forall x y z, z >A 0 -> x >A y -> x * z >A y * z.
End OrdRingType.
```

We implement the theory of order ring and setoid reflexive closure of this order ring.

```
Module OrdRingTheory (Export ORT : OrdRingType).
  Module Export RT := RingTheory R.
  Definition geA x y := x >A y \/ x =A= y.
  Lemma geA_refl : reflexive geA.
  Lemma geA_trans : transitive geA.
```

We define the boolean equality and checking its correctness

```
Definition beqA x y :=
   match eqA_dec x y with
     | left _ => true
     | right _ => false
   end.
Lemma beqA_ok : forall x y, beqA x y = true <-> x =A= y.
```

We define for non negative predicate

```
Notation not_neg := (fun z => z >=A 0).
Definition bnot_neg z := bgtA z 0 || beqA z 0.
Lemma bnot_neg_ok : forall z, bnot_neg z = true <-> z >=A 0.
```

After that, we proof some properties of this order ring. For instance:

```
Lemma geA_gtA_trans : forall x y z, x >=A y -> y >A z -> x >A z.
Lemma mul_gtA_0_compat : forall n m, n >A 0 -> m >A 0 -> n * m >A 0.
Lemma power_geA_0 : forall x n, x >=A 0 ->  x ^ n >=A 0.
```

We use the same technique for integer numbers and rational numbers with order ring.

Ordered rings over integer numbers $\mathbb{Z}$

```
Module IntOrdRing <: OrdRingType.
  Module Export R := IntRing.
  Module Export RT := RingTheory R.
  Require Import ZArith.
  Definition gtA := Zgt.
  Add Morphism gtA with signature eqA ==> eqA ==> iff as gtA_morph.
  Definition gtA_dec := Z_gt_dec.
  Definition bgtA x y :=
    match gtA_dec x y with
      |left _ => true
      |right _ => false
    end.
  Lemma bgtA_ok : forall x y, bgtA x y = true <-> (x > y)%Z.
  Lemma gtA_trans : transitive gtA.
  Lemma gtA_irrefl : irreflexive gtA.
  Lemma one_gtA_zero : (1 > 0)%Z.
  Lemma add_gtA_mono_r : forall x y z,(x > y)%Z -> (x + z > y + z)%Z.
  Lemma mul_gtA_mono_r : forall x y z,(z > 0)%Z -> (x > y)%Z
                           -> (x * z > y * z)%Z.
End IntOrdRing.
Module IntOrdRingTheory := OrdRingTheory IntOrdRing.
```

Ordered rings over rational numbers $\mathbb{Q}$

```
Module RatOrdRing <: OrdRingType.
   Module Export R := RatRing.
   Module Export RT := RingTheory R.
   Require Import QArith.
   Definition gtA x y := Qlt y x.
   Add Morphism gtA with signature eqA ==> eqA ==> iff as gtA_morph.
   Definition Q_gt_dec x y : {x > y} + {~ x > y}.
   Definition gtA_dec := Q_gt_dec.
   Definition bgtA x y :=
     match gtA_dec x y with
       |left _ => true
       |right _ => false
     end.
    Lemma bgtA_ok : forall x y, bgtA x y = true <-> x > y.
    Lemma gtA_trans : transitive gtA.
    Lemma gtA_irrefl : irreflexive gtA.
    Lemma one_gtA_zero : 1 > 0.
    Lemma add_gtA_mono_r : forall x y z, x > y -> x + z > y + z.
    Lemma mul_gtA_mono_r : forall x y z, z > 0 -> x > y -> x * z
End RatOrdRing.
Module RatOrdRingTheory := OrdRingTheory RatOrdRing.
```

6.3 Well-foundedness proof of the δ-ordering on $\mathbb{Q}^+$

Polynominals with *real* coefficients were proposed by Dershowitz [6] as an alternative to polynomials over the naturals in proofs of termination of rewriting. In fact, rewriting systems that can be proved polynomially terminating by using polynomial interpretations with (algebraic) real coefficients; however, the proof cannot be achived if polynomials only contain rational coefficients. Since the set of real numbers $\mathbb{R}$ furnished with the usual ordering $>_{\mathbb{R}}$ is not well-founded, a *subterm* property (i.e., $f(x_1, ..., x_i, ..., x_k) >_{\mathbb{R}} x_i$ for all k-ary symbols f, arguments1 $\leq i \leq k$, and $x_1, ..., x_k \in \mathbb{R}$) is explicitly required to ensure well-foundedness.

A new framework for proving termination of TRSs by using polynomials over the reals has recently been introduced in [5].

Given a positive real number $\delta \in \mathbb{R}_{>0}$, a well-founded and stable (strict) ordering $>_\delta$ on terms is defined as follows: for all $t, s \in T(F, V), t >_\delta s$ if and only if $[t] - [s] \geq_{\mathbb{R}} \delta$.

We want to prove that $\forall x, y \in \mathbb{Q}^+, x >_\delta y \Rightarrow f(x) > f(y)$. Let $x, y \in \mathbb{Q}^+$ such that $x >_\delta y$. By Euclidian division, there is t such that $x = f(x)\delta + t$ and $0 \leq t < \delta$. Similarly, there is u such that $y = f(y)\delta + u$ and $0 \leq u < \delta$. So, $x - y = (f(x) - f(y))\delta + (t - u)$ and $-delta < t - u < \delta$. Now, if $x >_\delta y$ then, by definition, $x - y = (f(x) - f(y))\delta + (t - u) \geq \delta$. So, $(f(x) - f(y))\delta \geq \delta - (t - u) > 0$ since $-\delta < t - u < \delta$. By dividing by $\delta > 0$, we get $f(x) - f(y) > 0$.

We now formalize it in Coq by giving two variables: delta with type $\mathbb{Q}$ of rational numbers, and delta_pos is a proof that $\delta > 0$. We define the rule gtA x y := x - y >= delta.

We have: $f(x) = \lfloor \frac{x}{\delta} \rfloor$ with definition f_Z x.

The definition f x is return the function x has type $\mathbb{N}$, f_Q x is the function x with type $\mathbb{Q}$.

```
Variable delta : Q.
Variable delta_pos : delta > 0.
Definition gtA x y :=  x - y >= delta.
Notation "x >A y" := (gtA x y).

Definition f (x : Q) : nat := Zabs_nat (f_Z x).
Definition f_Q (x : Q) : Q := inject_Z (f_Z x).
Definition f_Z (x : Q) : Z := Qfloor (Qdiv x delta).
```

The lemma poly_exp is proving: $\forall x \in \mathbb{Q}^+$, $\exists t \in \mathbb{Q}$, $x = f(x)\delta + t$ and $0 \leq t < \delta$.

```
Lemma poly_exp:  forall x, x >= 0 -> exists t,
  x == f_Q x * delta + t /\ 0 <= t /\ t < delta.

Lemma terms_cond : forall t u, 0 <= t /\ t < delta /\
  0 <= u /\ u < delta ->  - delta < t + - u /\ t + - u < delta.
```

The lemma wf_Q_N prove it is well-founded from the domain $(\mathbb{Q}^+, >_\delta)$ to $(\mathbb{N}, >_{\mathbb{N}})$ such that $x >^\delta_{\mathbb{Q}} y \Rightarrow f(x) >_{\mathbb{N}} f(y)$.

```
Lemma wf_Q_N : forall x y, x >= 0 /\ y >= 0 -> x>Ay -> (f(x)>f(y)).
```

We consider polynomials using nonnegative, *rational* coefficients and proved it is well-founded by using the lemma wf_Q_N above.

```
Definition gtA_not_neg x y := gtA x y /\ x >= 0 /\ y >= 0.
Theorem well_founded_gtA_not_neg: well_founded (transp gtA_not_neg).
```

6.4 Improved monotony criterion

In this subsection we present an improved definition of weak and strong monotony criterion, also checked the correctness of its condition by using the boolean function.

The definition of monomial x_i^k:

```
Fixpoint mxi n : forall i, i < n -> nat -> monom n :=
  match n as n return forall i, i < n -> nat -> monom n with
    | 0 => fun i h _ => False_rec (monom 0) (lt_n_0 i h)
    | S n' => fun i =>
      match i as i return lt i (S n') -> nat -> monom (S n') with
        | 0 => fun _ k => Vcons k (mone n')
        | S _ => fun h k => Vcons 0 (mxi (lt_S_n h) k)
      end
  end.
```

Improve the monotony criterion which means that the monotonicity of a polynomial P of n indeterminates is ensured by requiring that, for all $i < n$, the coefficient of x_i^k in P is (strictly) positive.

```
Definition pstrong_monotone2 n (p : poly n) := pweak_monotone p
 /\ forall i (H : i < n), exists k, coef (mxi H k) p >A 0.
```

We have nat_lt is a set of natural number smaller than n. nats_lt is a list of natural numbers strictly smaller than n with proofs. nfirst is a list from $(n-1, ..., 0)$.

```
Record nat_lt (n : nat) : Type :=
  mk_nat_lt { val : nat; prf : val < n }.

Definition nats_lt : forall n : nat, list (nat_lt n) := ...
  (* list of numbers smaller than n with the proofs *)

Fixpoint nfirst n :=
  match n with
    | 0 => nil
    | S k => k :: nfirst k
  end.
```

The boolean function of weak monotone checking the absolute positiveness such that a polynomial is absolutely positive if all its coefficients are non-negative. We give a variable *kmax* is a natural number such that $kmax > k$ testing that each monomial x_i^k is (strictly) positive.

```
 Definition bpweak_monotone n (p : poly n):= bcoef_not_neg p.
 Definition bpweak_monotone_ok n (p : poly n):= bcoef_not_neg_ok p.

 Variable kmax : nat.
 Definition bpstrong_monotone2 n (p : poly n) :=
  bcoef_not_neg p
  && existsb
       (fun k =>
          forallb
            (fun x => bgtA (coef (mxi (prf x) k) p) 0)
            (nats_lt n))
     (nfirst kmax).

 Lemma bpstrong_monotone2_ok : forall n (p : poly n),
   bpstrong_monotone2 p = true -> pstrong_monotone2 p.
```

existsb is a boolean function find whether a boolean function can be satisfied by an elements of the list.

```
Fixpoint existsb (l:list A) : bool :=
   match l with
     | nil => false
     | a::l => f a || existsb l
   end.
```

forallb also a boolean function find whether a boolean function is satisfied by all the elements of a lists.

```
Fixpoint forallb (l:list A) : bool :=
   match l with
     | nil => true
     | a::l => f a && forallb l
   end.
```

6.5 Polynomials of degree 2 with negative coefficients

Next we proof the well-defined of generic quadratic parametric polynomial function:

$$f_{\mathbb{N}}(x_1, ..., x_n) = c + \sum_{i=1}^{n} bx_i + \sum_{i=1}^{n} \sum_{j=i}^{n} a_{ij}x_{ij} \in \mathbb{Z}[x_1, ..., x_n].$$

Theorem 8 [7] The function $f_{\mathbb{N}}$ is strictly (weakly) monotone and well-defined iff $c \geq 0$, $a_{ij} \geq 0$ and $b > -a_{ii}$ ($b \geq -a_{ii}$) for all $1 \leq i \leq j \leq n$.

Proof. [7] By induction hypothesis, we have the first case: $f_{\mathbb{N}}(0,...,0) \geq 0$ iff $c \geq 0$ $(\forall x_1,..,x_n \in \mathbb{N})$;
Induction case:

$$f_{\mathbb{N}}(x_1,...,x_i+1,...,x_n) \geq f_{\mathbb{N}}(x_1,...,x_i,...,x_n)$$

becomes
$b(x_i{+}1){+}a_{ii}(x_i{+}1)^2{+}\sum_{1\leq j\leq n, j\neq i} a_{ij}(x_i{+}1)x_j > bx_i{+}a_{ii}x_i^2{+}\sum_{1\leq j\leq n, j\neq i} a_{ij}x_ix_j$
which simplifies to

$$b + 2a_{ii}x_i + a_{ii} + \textstyle\sum_{1\leq j\leq n, j\neq i} a_{ij}x_j > 0$$

This formula holds for all $x_1,...,x_n \in \mathbb{N}$ iff $a_{ij} \geq 0 (j \neq i)$ and $2a_{ii}x_i{+}a_{ii}{+}b > 0$ for all $x_i \in \mathbb{N}$ iff $a_{ii} \geq 0$ and $b > -a_{ii}$. Altogether, this proves claim for strict monotonicity; for weak monotonicity we just have to replace $>$ by $\geq$ in the above calculation.

□

Corollary 9 The function $f_{\mathbb{N}}(x) = ax^2 + bx + c$ is strictly (weakly) monotone and well-defined iff $a \geq 0$, $c \geq 0$, and $b > -a$ $(b \geq -a)$.

Hence, in a quadratic polynomial all coefficients must be non-negative except the coefficients of the linear monomials.

Now we formalize it in `Coq`. The definition of degree and the condition saying that if the degree of coefficient is greater or equal than 3, its coefficient is equal to 0.

```
Fixpoint degree n (v : monom n) {struct v} : nat :=
  match v with
    | Vnil => 0
    | Vcons k _ v => (k + degree v)%nat
  end.

Variable hyp : forall m : monom n, degree m >= 3 -> coef m p =A= 0.
```

We have *mxij* is a monomial multiplication x_ix_j. *a*,*b*,*c* are the coefficients of polynomial.

```
Definition mxij i (hi: i<n) j (hj: j<n):=
  mmult (mxi hi 1) (mxi hj 1).
Definition a i (hi: i<n) j (hj: j<n) := coef (mxij hi hj) p.
Definition b i (hi: i<n) := coef (mxi hi 1) p.
Definition c := coef (mone n) p.
```

From theorem 5 we have the definition of the function $f_{\mathbb{N}}$ is strictly (weakly) monotone and well-defined.

```
Definition monotone :=
    c >=A 0
    /\ forall j (hj: j<n), b hj >=A - a hj hj
      /\ forall i (hi: i<n) j (hj: j<n), a hi hj >=A 0.

Definition strict_monotone :=
   c >=A 0
    /\ forall j (hj: j<n), b hj >A - a hj hj
      /\ forall i j (hi: i<n) (hj: j<n), a hi hj >=A 0.
```

We give the definition checking the well-defined and correctness. The boolean function saying if the coefficient b of monomial x_i is greater than equal the negative coefficient $-a$ of quadratic polynomial or not, and a proof that this function is correct.

P is a condition of coefficient x_i in linear monomials.

```
Definition P j := forall (hj: j<n), b hj  >=A - a hj hj.

Definition bP j := match lt_ge_dec j n with
                       | left hj => bgeA (b hj) (- a hj hj)
                       | _ => true
                      end.
Lemma bP_ok : forall j, bP j = true <-> P j.
```

P' is the definition of strictly condition. bP' is the boolean function of P', we also checking the correctness of this definition.

```
Definition P' j := forall (hj: j<n), b hj >A - a hj hj.
Definition bP' j := match lt_ge_dec j n with
                        | left hj => bgtA (b hj) (- a hj hj)
                        | _ => true
                      end.

Lemma bP'_ok : forall j, bP' j = true <-> P' j.
```

The coefficient of $x_i x_j$ in quadratic polynomial has two conditions, we call the first one is R such that for all $j < n$, $i < n$ we have: $a_{ij} >= 0$. bR is a boolean function of R and after that we proof the correct of definition bR is true for all j.

```
Definition R j := forall (hi: i<n) (hj: j<n), a hi hj >=A 0.
Definition bR j :=
    match lt_ge_dec i n, lt_ge_dec j n with
      | left hi, left hj => bnot_neg (a hi hj)
      | _, _ => true
    end.

Lemma bR_ok : forall j, bR j = true <-> R j.
```

The definition forall_lt is checking for all $i < n$. Such that P is satisfied for all $i < n$ and bP is true for all i.

```
 Variables (P: nat -> Prop) (bP: nat -> bool)
    (bP_ok: forall i, bP i = true <-> P i).

 Definition forall_lt n := forall i, i<n -> P i.
 Fixpoint bforall_lt n :=
     match n with
       | O => true
       | S n' => bP n' && bforall_lt n'
     end.
```

Now we define the last condition for the quadratic polynomial, for all $i < n$, $a_{ij} >= 0$.

```
 Definition Q i := forall_lt (R i) n.
 Definition bQ i := bforall_lt (bR i) n.
```

We generalize the monotone' using the function $forall\_lt$ and proving that it is equality with the definition of monotone.

After that, we give the definition of boolean functions of monotone and strictly monotone and proof the correctness of this definition.

```
Definition monotone' :=
  not_neg c /\ forall_lt P n /\ forall_lt Q n.

Lemma monotone_eq' : monotone <-> monotone'.

Definition bmonotone :=
  bnot_neg c && bforall_lt bP n && bforall_lt bQ n.

Definition bstrict_monotone :=
  bnot_neg c && bforall_lt bP' n && bforall_lt bQ n.

Lemma bmonotone_ok : bmonotone = true <-> monotone.

Lemma bstrict_monotone_ok :
  bstrict_monotone = true <-> strict_monotone.
```

7 Conclusion

In this report we presented an approach to certification of termination proofs, by using interpretation play an important role in termination. We showed how to formalization in generic interface for rings and orderings in polynomial interpretation. We also showed how this formalization is the well foundedness with δ-ordering on $\mathbb{Q}^+$.

Improved the monotony criterion by checking the coefficient of monomial x_i^k is (strictly) positive. Analyzed polynomials is a bound on the degree 2 or quadratic polynomials with negative coefficients and checking the correctness of the parameters by testing the equality of the corresponding boolean functions to true (reflexivity proof).

Future worke is design termination certificates and develope procedures to automatically and efficiently verify their correctness with the highest confidence. To improve this, we need to certify Rainbow[7] (a termation certificate checker) itself in `Coq` by formalizing the certificates themselves, defining a boolean function saying if a certificate is correct or not, and proving that this function is correct, that is, that one can indeed build a termination proof if this function returns true. Then, by using Coq's extraction mechanism [8], we can get an efficient standalone termination certificate checker.

Another way to improve the verification of termination certificates, not only in speed but also in the number of termination criteria that can be handled, is to prove the correctness of each node separatly, possibly with different verification tools.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates, July 2010.

[4] Sébastien Hinderer. Certification of Termination Proofs Using Polynomial Interpretations. In *17th European Summer School in Logic, Language and Information - ESSLLI '05*, Edimbourg/Grande Bretagne, 08 2005.

[5] Salvador Lucas. Polynomials over the reals in proof of termination. In *In Proc. 7th Internation Workshop on Termination, Technical Report AIB-2004-07, RWTH Aachen*, pages 39–42, 2004.

[6] Dershowitz Nachum. A Note on Simplification Orderings, Apr. 1979. Report DCS-4-79-986.

[7] Friedrich Neurauter, Aart Middeldorp, and Harald Zankl. Monotonicity criteria for polynomial interpretations over the naturals. In *IJCAR*, pages 502–517, 2010.

[7] http://color.inria.fr/#rainbow

[8] C. Paulin-Mohring. Extracting omega's programs from proofs in the calculus of constructions's programs from proofs in the calculus of constructions. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 89–104, New York, NY, USA, 1989. ACM.

[9] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.