Cylindrical Algebraic Decomposition in Coq

MAP 2010 - Logroño 13-16 November 2010

Assia Mahboubi

INRIA Microsoft Research Joint Centre (France)
INRIA Saclay – Île-de-France
École Polytechnique, Palaiseau

November 11th 2010

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

What we have seen so far

We have investigated two levels of formalization:

- A formal representation of logical objects:
  - A formalization of the first order theory of discrete real closed fields;
  - A sufficient condition to obtain full quantifier elimination;

    Eliminate $\exists$ in $\exists x, \bigwedge_{i=1}^{n} L_i$
- A formal representation of geometrical objects:
  - A formalization of models of the real closed field structures (as records)
  - A geometrical (constructive) proof that the projection of semi-algebraic sets is a semi-algebraic set

    $\pi_{|x_{k+1}} \{x \in R^{k+1} \mid P(x) = 0 \wedge \bigwedge_{Q \in \mathcal{Q}} Q(x) > 0\}$ is a semi-algebraic set.

What we have seen so far

This in fact covers three different levels:

- The programmer (computer algebra)
- The Coq user (formalizing correctness of computer algebra)
- The Coq logician user (formalizing theorems of logic)

The programmer

These are constructive proofs, we can write programs. For instance:

- `l_coef(p : poly R): R := ...`
- `count_sign_changes(l : seq R): nat := ...`
- `signed_prem(p q : poly R): seq poly R := ...`
-
  ```
  tarski_query (p q : poly R) : nat :=
    count_sign_changes
      (map lcoef (signed_prem p (p^`() * q)))
  ```
-
  ```
  test_sas_empty1 (p q : poly R) : bool :=
     (tarski_query p q^2) + (tarski_query p q) > 0
  ```

The Coq user, interested in real algebraic geometry

These are constructive proofs, we can write program (in Coq):

```
Definition test_sas_empty1 (p q : {poly R}) : bool :=
   (tarski_query p q^2) + (tarski_query p q) > 0
```

But moreover we want to formalize the proof that:

```
Theorem test_sas_empty1_correct : forall p q : {poly R},
   p != 0 ->
   (test_sas_empty1 p q) > 0
      <->
   (exists x, p.[x] = 0 /\ q.[x] > 0).
```

The other Coq user, interested in formal proofs in logic

We want a running quantifier elimination algorithm:

- A program:

```
Fixpoint quantifier_elim :
  formula term -> formula term := ...
```

- And formal proofs that:

```
Lemma q_free_quantifier_elim : forall f,
   q_free (quantifier_elim f).

Lemma quantifier_elim_correct : forall (R_rcf : rcf),
  forall (f : formula term)(ctx : seq (R R_rcf)),
  (holds ctx f) <-> (holds ctx (quantifier_elim f)).
```

Sharing the efforts

- Can the programmer rely on mathematics textbooks?

  Yes, if they are reasonably written (for that purpose).
- Can the Coq user rely on the programmer?

  Yes, if the programmer uses a pure functional language.
- Can the logician Coq user rely on the geometer Coq user?

  Unclear at this stage.

Can the logician Coq user rely on the geometer Coq user?

- Is it “easy” to understand the semi-algebraic object described by the input formula?

Can the logician Coq user rely on the geometer Coq user?

- Is it “easy” to understand the semi-algebraic object described by the input formula?

Yes, very easy.

Can the logician Coq user rely on the geometer Coq user?

- Is it “easy” to understand the semi-algebraic object described by the input formula?

Yes, very easy.

- Is it “easy” to read the expected, quantifier free formula from the computer algebra programs?

?

Can the logician Coq user rely on the geometer Coq user?

- Is it “easy” to understand the semi-algebraic object described by the input formula?

  Yes, very easy.

- Is it “easy” to read the expected, quantifier free formula from the computer algebra programs?

  ?

- Is it “easy” to prove that this formula is correct with respect to the initial one?

  ?

The Coq programs

- `l_coef(p : poly R): R := ...`
- `count_sign_changes(l : seq R): nat := ...`
- `signed_prem(p q : poly R): seq poly R := ...`
-
  ```
  tarski_query (p q : poly R) : nat :=
    count_sign_changes
      (map lcoef (signed_prem p (p^`() * q)))
  ```
-
  ```
  test_sas_empty1 (p q : poly R) : bool :=
     (tarski_query p q^2) + (tarski_query p q) > 0
  ```

Execution of the program for $\{x \in R | \alpha x^2 + \beta x + \gamma = 0\}$

From programs to formulas

- The expected formula should collect the conditions leading to the desired result along all the successful paths.
- From the code as such, it might well be difficult.
- We need to expose (more) the control over the execution flow.

Concrete polynomials

- Univariate polynomials are represented by lists of coefficients.
- We only manipulate polynomials in normal form:
  - The empty list represents the zero polynomial.
  - The head of the list is the constant coefficient.
  - A non empty list has a head non zero element.

Example of program

- A program computing the leading coefficient:

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

- A program testing that the leading coefficient is positive:

```
Definition test (p : {poly R}) : bool :=
   lcoef p > 0.
```

- What is the counterpart at the formula level?

Terms, the parameterized ring signature

```
Inductive term (R : Type) : Type :=
 | Var of nat
 | Const of R
 | NatConst of nat
 | Add of term & term
 | Opp of term
 | Mul of term & term.
```

- An atom is a term compared to zero (after reduction).
- Terms are polynomial expressions in their free variables.

First order theory, again with parameters

```
Inductive formula (R : Type) : Type :=
| Equal of (term R) & (term R)
| Leq of (term R) & (term R)
| Lt of (term R) & (term R)
| trueF : formula R
| falseF : formula R
| Not of formula
| And of formula & formula
| Or of formula & formula
| Implies of formula & formula
| Exists of nat & formula
| Forall of nat & formula.
```

Abstract polynomials

Consider the formula with a single existential quantifier:

$$\exists x, \alpha x^2 + (\beta x + 1) + \alpha\gamma = 0$$

- The atom is a sign condition on the term $\alpha x^2 + \beta x + \gamma$;
- The single quantifier binds the variable x;
- The term in the atom should be understood as a polynomial, element of $R[\alpha, \beta, \gamma][x]$

Abstract polynomials

Consider the formula with a single existential quantifier:

$$\exists x, \alpha x^2 + (\beta + 1)x + \alpha\gamma = 0$$

- The term embedded in such an atom can be seen as an abstract univariate polynomial, with abstract polynomial coefficients.
- An abstract univariate polynomial is represented by lists of terms.
  $[\alpha, \beta + 1, \alpha\gamma]$: `(seq (term R))`
- An abstract coefficient is only a term.

Abstract polynomials

- From a `(t : term)` in an atom, and the name i of the variable bound by the existential, we can extract the abstract univariate polynomial in the variable x_i thanks to the function:

```
Fixpoint abstrX (i : nat) (t : term R) : (seq term) :=
    ...
```

- An abstract univariate polynomial can be interpreted to a univariate polynomial given a context:

```
Fixpoint eval_polyF (e : seq R) (ap : (seq term)) : {
    poly R} :=
  match ap with
  |c :: qf => (eval_polyF e qf)*'X + (eval e c)
  |[::] => 0
end.
```

- We want the diagram to commute.

Inside out

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

Inside out

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

Inside out

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

Fixpoint cps_lcoef                  (p : {poly R}) :      :=




Definition cps_test (p : {poly R}) : bool :=
```

Inside out

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

Fixpoint cps_lcoef                    (p : {poly R}) :       :=
  match p with
  | [::] →
  | c :: q →
  end.
Definition cps_test (p : {poly R}) : bool :=
```

Inside out

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

Fixpoint cps_lcoef (k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] →
  | c :: q →
  end.
Definition cps_test (p : {poly R}) : bool :=
```

Inside out

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

Fixpoint cps_lcoef (k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q →
  end.
Definition cps_test (p : {poly R}) : bool :=
```

Inside out

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

Fixpoint cps_lcoef (k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q →  cps_lcoef                                          q
  end.
Definition cps_test (p : {poly R}) : bool :=
```

Inside out

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

Fixpoint cps_lcoef (k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q →  cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.
Definition cps_test (p : {poly R}) : bool :=
```

Inside out

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

Fixpoint cps_lcoef (k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q →  cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.
Definition cps_test (p : {poly R}) : bool :=
cps_lcoef                                     p
```

Inside out

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

Fixpoint cps_lcoef (k : R → bool) (p : {poly R}) : bool :=
   match p with
   | [::] → (k 0)
   | c :: q →  cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
   end.
Definition cps_test (p : {poly R}) : bool :=
cps_lcoef (fun n ⇒ if n > 0 then true else false) p
```

What happened in this transformation?

Consider the emptiness test for one dimensional basic semi-algebraic sets:

$$\{x \in R \mid P(x) = 0 \wedge \bigwedge_{Q \in \mathcal{Q}} Q(x) > 0\}$$

- It consists in assembling sign tests on polynomial expressions in the coefficients of P and the Qs:

$$c(f(p_1, \dots, p_k, q_{11}, \dots, q_{1,k_1}, \dots, q_{1,k_n}))$$

- These tests are the nodes in the tree of execution.
- For test on such a polynomial expression we can abstract over the control operation by:
  - Programming a CPS version *cps_f* of f
  - Giving the continuation k_c as an argument to f

Continuation passing style

- This is not (meant to be) code obfuscation.
- We have exposed the control operations by the mean of a continuation.
- This version of the code is ready to be translated at the formula level:
    - By turning boolean outputs into formulas outputs
    - By turning polynomials and coefficients into terms
- Remark : we can define a branching formula:
  Definition ifF (condF thenF elseF : formula R) : formula R := ((condF ∧ thenF) ∨ ((condF) ∧ elseF)).

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
   match p with
   | [::] → (k 0)
   | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
   end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
   match p with
   | [::] → (k 0)
   | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
   end.
```

```
Fixpoint cps_lcoefF
(k :          →              ) (p   :                 )) :              :=
   match p   with
   | [::] →
   | c :: q → cps_lcoefF                                                q
   end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.

Fixpoint cps_lcoefF
(k :        →               ) (pF :                 )) :                :=
  match pF with
  | [::] →
  | c :: q → cps_lcoefF                                                   q
  end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.

Fixpoint cps_lcoefF
(k :       →             ) (pF : (seq (term R))) :             :=
  match pF with
  | [::] →
  | c :: q → cps_lcoefF                                          q
  end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
   match p with
   | [::] → (k 0)
   | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
   end.
```

```
Fixpoint cps_lcoefF
(k : term R →                ) (pF : (seq (term R))) :               :=
   match pF with
   | [::] →
   | c :: q → cps_lcoefF                                              q
   end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.

Fixpoint cps_lcoefF
(k : term R → (formula R)) (pF : (seq (term R))) : (formula R) :=
  match pF with
  | [::] →
  | c :: q → cps_lcoefF                                            q
  end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.

Fixpoint cps_lcoefF
(k : term R → (formula R)) (pF : (seq (term R))) : (formula R) :=
  match pF with
  | [::] → (k (Const 0))
  | c :: q → cps_lcoefF                                              q
  end.
```

Formula level programs

```
Fixpoint cps_lcoef
(k : R → bool) (p : {poly R}) : bool :=
  match p with
  | [::] → (k 0)
  | c :: q → cps_lcoef (fun l ⇒ if (q == 0) then (k c) else (k l)) q
  end.

Fixpoint cps_lcoefF
(k : term R → (formula R)) (pF : (seq (term R))) : (formula R) :=
  match pF with
  | [::] → (k (Const 0))
  | c :: q → cps_lcoefF (fun l ⇒ ifF (Equal l (Const 0)) (k c) (k l)) q
  end.
```

Formula level programs

```
Definition cps_test (p : {poly R}) : bool :=
cps_lcoef
(fun n ⇒ if n > 0 then true else false)
p
```

Formula level programs

```
Definition cps_test (p : {poly R}) : bool :=
cps_lcoef
(fun n ⇒ if n > 0 then true else false)
p

Definition cps_testF (p : term R) : formula R :=
cps_lcoefF
(fun n ⇒ ifF (Lt (Const n) (Const 0)) trueF falseF)
p
```

Formula level programs

```
Definition cps_test (p : {poly R}) : bool :=
cps_lcoef
(fun n ⇒ if n > 0 then true else false)
p

Definition cps_testF (p : term R) : formula R :=
cps_lcoefF
(fun n ⇒ ifF (Lt (Const n) (Const 0)) trueF falseF)
p

Definition cps_cps_testF
(k : term R → formula R) (p : term R) : formula R :=
cps_lcoefF
(fun n ⇒ k (Lt (Const n) (Const 0)) p
p
```

What happened in this transformation?

Consider an abstract polynomial `pF : seq (term R))`, extracted from a basic formula:

- The concrete shape of this polynomial depends on the values instantiating the parameters.

`(eval_polyF e pF)` denoted `[pF]_e`

What happened in this transformation?

Consider an abstract polynomial `pF : seq (term R))`, extracted from a basic formula:

- The concrete shape of this polynomial depends on the values instantiating the parameters.

  `(eval_polyF e pF)` denoted `[pF]_e`
- Any polynomial function `f` has a formula CPS counterpart `fF`.

  `cps_lcoef` and `cps_lcoefF`

What happened in this transformation?

Consider an abstract polynomial `pF : seq (term R))`, extracted from a basic formula:

- The concrete shape of this polynomial depends on the values instantiating the parameters.

  `(eval_polyF e pF)` denoted `[pF]_e`
- Any polynomial function `f` has a formula CPS counterpart `fF`.

  `lcoef` and `cps_lcoefF`
- Any test `c` on such a polynomial expression has a formula CPS counterpart `(fF kc)`.

  `cps_testF`

Correctness as observational equivalence

Now we have commutation:

```
Lemma cps_lcoefFP : forall k pF e, acceptable_cont k ->
  qf_sat e (cps_lcoefF k pF)
  =
  qf_sat e (k (Const (lcoef [pF]_e))).
```

A generic and uniform process

- Program the concrete emptiness test for polynomials in $R[X]$;
- For every elementary program used in the previous phase:
  - Turn the concrete program into a CPS-formula one;
  - State the lemma corresponding to its correctness with respect to the concrete program;
  - Prove this lemma by executing symbolically the code of the concrete program in the proof.

Gluing the programs, and the proofs

- Combine the CPS-formula programs in the same way they are combined in the concrete emptiness test program;
- The quantifier elimination procedure of a single $\exists$ follows.
- Combine the CPS-formula correctness lemmas accordingly.
- The correctness proof follows.

Formalized quantifier elimination for discrete RCF

We have defined in Coq:

- The first order language of ordered fields
- The models of the theory of discrete real closed fields

Formalized quantifier elimination for discrete RCF

We have programed in Coq:

- A reduction of the full first order theory decidability to the elimination of a single ∃;
- An emptiness test for semi-algebraic sets;
- A transformation of this test into a procedure elimination a single ∃ by a syntactic process;

Formalized quantifier elimination for discrete RCF

We have formally proved in Coq:

- The reduction of decidability to quantifier elimination
- The reduction of full quantifier elimination to weak;
- The correctness of the emptiness test;
- The correctness of the weak quantifier elimination.

Formalized quantifier elimination for discrete RCF

We have formally proved in Coq:

- The reduction of decidability to quantifier elimination
- The reduction of full quantifier elimination to weak;
- The correctness of the emptiness test;
- The correctness of the weak quantifier elimination.

Disclaimer:

- This is in fact work in progress with C. Cohen.
- The analogue work on algebraically closed fields is completed.

Decidability, effectiveness

We have not addressed tractable programs so far: the complexity of this algorithm is far from elementary.
It is in fact comparable to the one of the original proof of Tarski.

Decision methods in discrete real closed fields

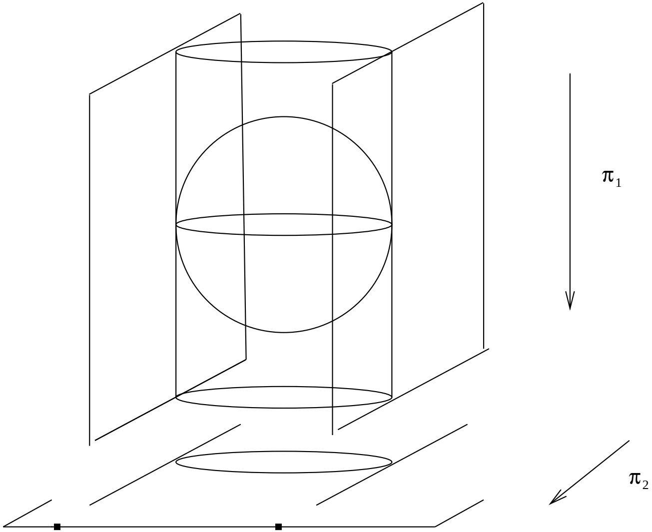
- Real closed field theory is decidable (Tarski, 1948)
- Hörmander method (Hörmander 1983 - Cohen 1969)
- Cylindrical Algebraic Decomposition algorithm (Collins 1975)

CAD in a nutshell: general setting

- Input: A finite family $\mathcal{P} \subset \mathbb{Q}[X_1, \ldots, X_n]$ of polynomials
- Output: A finite partition of $\mathbb{R}^n$ into cylindrical cells over which each element of $\mathcal{P}$ has a constant sign.

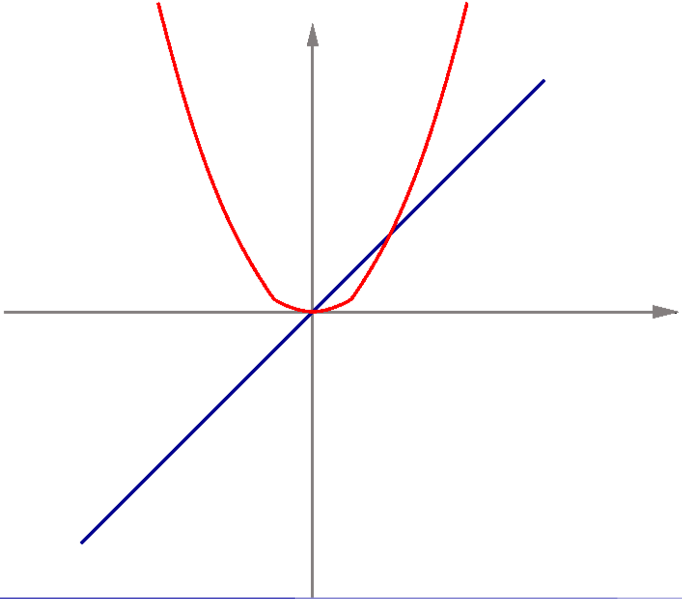
Warning: cylindrical decomposition does not entail decidability
(cf. Michel's tutorial) ...

Example: $X^2 + Y^2 + Z^2 - 1$



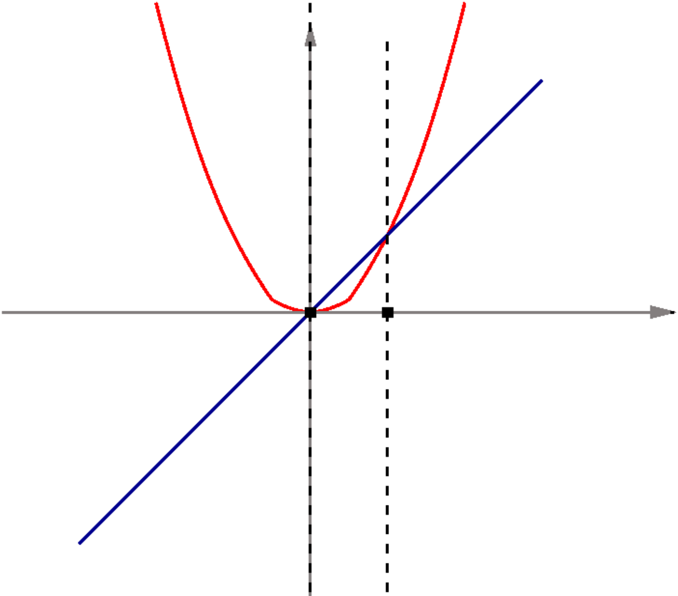
Prove $\exists x \exists y \quad y - x^2 = 0 \wedge x - y = 0$

Extract polynomials: $P_1 = y - x^2$, $P_2 = y - x$



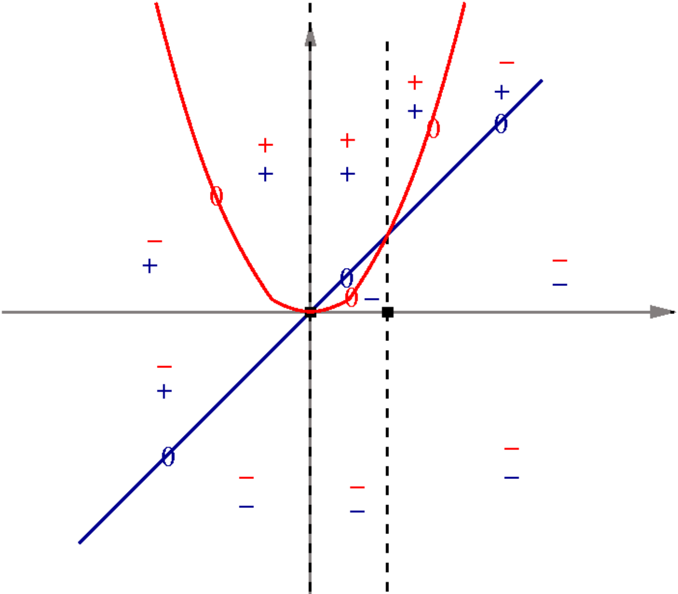
Prove $\exists x \exists y \quad y - x^2 = 0 \wedge x - y = 0$

Projection along y.



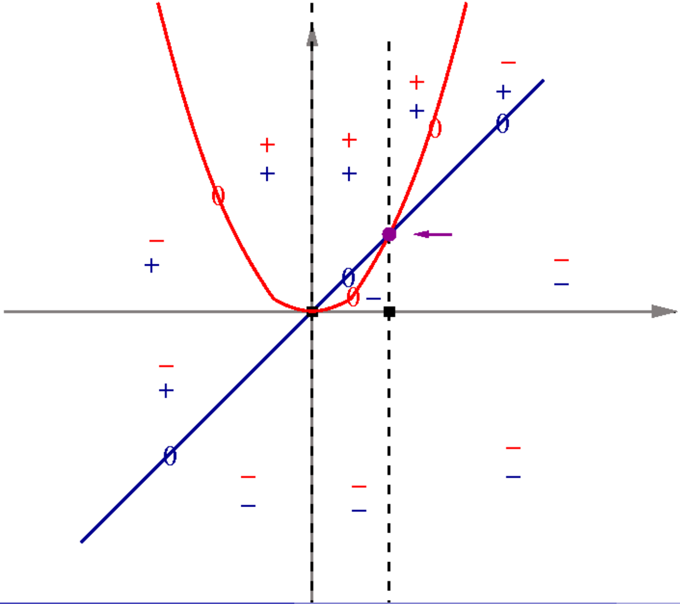
Prove $\exists x \exists y \quad y - x^2 = 0 \wedge x - y = 0$

Lifting phase and sign computation.



Prove $\exists x \exists y \quad y - x^2 = 0 \wedge x - y = 0$

Solution search



CAD in a nutshell

$$\begin{tikzcd}
\mathbb{R}[X_1, \ldots, X_n] & \mathbb{R}[X_1, \ldots, X_{n-1}] \\
\mathcal{P} = P_1, \ldots, P_s \arrow[r, "\text{projection}"] & \mathcal{Q} = Q_1, \ldots, Q_t \\
\text{CAD and signs for } \mathcal{P} & \text{CAD and signs for } \mathcal{Q} \arrow[l, "\text{lifting}"'] \\
\mathbb{R}^n & \mathbb{R}^{n-1}
\end{tikzcd}$$