Journées Nationales de Calcul Formel

Rencontre organisée par :
Guillaume Chèze, Thomas Cluzeau, Grégoire Lecerf et Clément Pernet

2011

Loïc Pottier

Preuves formelles automatiques et calcul formel

Vol. 2, n° 1 (2011), Cours n° III, p. 1-25.

<http://ccirm.cedram.org/item?id=CCIRM_2011__2_1_A3_0>

Centre international de rencontres mathématiques
U.M.S. 822 C.N.R.S./S.M.F.
Luminy (Marseille) France

Les cours du C.I.R.M.
Vol. **2** n° 1 (2011) 1-25
Cours n° III

Preuves formelles automatiques et calcul formel

Loïc Pottier

Table des matières

1. Introduction

Ce cours a pour but de montrer que l'on peut utiliser un assistant de preuve pour faire du calcul formel à la fois certifié et efficace. Il est en 2 parties.

La première est une initiation à l'assistant de preuve Coq, sur un exemple avec les polynômes. On définit des algorithmes et on en prouve des propriétés.

La seconde partie décrit une expérience d'intégration d'une procédure de décision à Coq et de son application à la preuve automatique de théorèmes de géométrie. Elle est directement tirée d'un article écrit avec Benjamin Grégoire et Laurent Théry [12].

Cours professé lors de la rencontre « Journées Nationales de Calcul Formel » organisée par Guillaume Chèze, Thomas Cluzeau, Grégoire Lecerf et Clément Pernet. 14-18 novembre 2011, C.I.R.M. (Luminy).

2. L'assistant à la preuve Coq

Le système Coq est un assistant à la preuve, c'est-à-dire qu'il fournit des outils pour que l'utilisateur construise une preuve d'un théorème qu'il a énoncé. Le système vérifie ensuite formellement la preuve et s'il réussit, la preuve est acceptée et enregistrée. Il dispose de procédures de décisions qui construisent automatiquement des preuves dans différents domaines (logique, algèbre). Il dispose aussi d'un langage de programmation fonctionnel, similaire à ocaml. Ce langage est compilé, bientôt nativement. Le langage du système est le *Calcul des constructions inductives*, une théorie des types généralisant les travaux de Russell, Church, Martin-Löf et Girard, due à Coquand, Huet, Paulin et Werner. Ce langage est adpaté aux mathématiques, et un des derniers exemples est dû à Voevodsky.

Faisons tout de suite un peu de calcul formel avec Coq. Idéalement, ce qui suit est à lire en même temps que l'on exécute les commandes. La place manque pour afficher tous les résultats qu'affiche Coq, mais déjà sans jouer avec Coq en parallèle, j'espère que cela donne une idée de ce qu'il est possible de faire en débutant. Les quelques résultats affichés par Coq qu'on donne ici seront précédés de "`Coq :`".

Le fichier complet du code est sur le Web ici :
`http ://www-sop.inria.fr/marelle/Loic.Pottier/jncf11.v`.

2.0.1. *Polynômes de $Z[X]$.* On charge l'arithmétique entière.

```
Require Export ZArith ZArith_dec.
Open Scope Z_scope.
```

en Coq, ne pas oublier le point à la fin des phrases, et la majuscule au début, du moins pour les commandes.

2.0.2. *Définition.* Le type des polynômes en représentation dense : liste des coefficients de chaque degré. 2 constructeurs :

- Pc prend un entier de Z en argument, et représente un polynôme constant
- PX prend un polynôme p et un entier a, et représente X*p + aPX

en ocaml on aurait écrit

```
type pol =
  | Pc of int
  | PX of pol * int
```

en Coq cela devient :

```
Inductive Pol:Type:=
 | Pc: Z -> Pol
 | PX: Pol -> Z -> Pol.
```

le polynôme X + 2 : (la commande Check donne le type)

```
Check PX (Pc 1) 2.
Coq:
PX (Pc 1) 2 : Pol
```

on allège les notations : la coercion permet de voir un entier comme un polynôme :

```
Coercion Pc: Z >-> Pol.
```

et on étend la grammaire :

```
Notation "'X' * ( x ) + y" := (PX x y) (at level 70).
```

testons :

```
Check X * (3)+ 2.
Coq:
X  * (3)+ 2 : Pol
```

parfois, il y a beaucoup de coercions et les choses sont trop ambigües, on ne comprend plus rien, dans ce cas :

```
Unset Printing Notations.
Set Printing Coercions.
Check X * (3)+ 2.
```

```
Coq:
PX (Pc (Zpos (xI xH))) (Zpos (xO xH)) : Pol
```

apparaissent les coercions `Pc`, mais aussi la représentation interne des entiers, codés comme suites de bits `xI`, `x0`. Remettons les notations et les coercions.

```
Set Printing Notations.
Unset Printing Coercions.
```

2.0.3. *Addition.* Programmons l'addition, par une fonction récursive : `{struct p}` indique que la récurrence porte sur p

```
Section addition.

Reserved Infix ".+" (at level 50, left associativity).

Fixpoint plusP1(p q:Pol){struct p}:Pol :=
  match p,q with
   | Pc a, Pc b => a + b
   | Pc a, X*(q1)+b => X*(q1)+(a + b)
   | X*(p1)+a, Pc b => X*(p1)+(a + b)
   | X*(p1)+a, X*(q1)+b => X*(p1 .+ q1) + (a + b)
  end
where "p .+ q" := (plusP1 p q).
```

calculons :

```
Eval compute in (X * (3)+ 2) .+ (X * (-3)+ -2).
Coq:
     = X  * (0)+ 0
     : Pol
```

ah ! zut... un coefficient nul en tête d'un polynôme, ca fait désordre !

```
End addition.
```

modifions un peu la construction X*(p)+a en traitant le cas où p est visiblement nul :

```
Definition px(p:Pol)(a:Z):Pol:=
  match p with
   | Pc 0 => Pc a
   | _ => PX p a
  end.
```

nouvelle version de l'addition, où l'on évite de mettre un coefficent nul au plus haut degré ; on démontrera plus tard que si p et q n'ont pas de coefficient nul au plus haut degré , alors il en est de même pour p+q

```
Reserved Infix ".+" (at level 50, left associativity).

Fixpoint plusP(p q:Pol){struct p}:Pol :=
  match p,q with
   | Pc a, Pc b => Pc (a + b)
   | Pc a, PX q1 b => PX q1 (a + b)
   | PX p1 a, Pc b => PX p1 (a + b)
   | PX p1 a, PX q1 b => px (p1 .+ q1) (a + b)
  end
where "p .+ q" := (plusP p q).
```

ah, là c'est bon :-)

```
Eval compute in (X * (3)+ 2)  .+  (X * (-3)+ -2).
Coq:
     = Pc 0
     : Pol
```

2.0.4. *Egalité.* Maintenant définissons l'égalité des polynômes, modulo les coefficients nuls. On aura besoin de définir la nullité d'un polynôme. On le fait avec un prédicat inductif : c'est le plus petit prédicat qui satisfait les axiomes de sa définition : nulP0 et nulPX

```
Inductive nulP:Pol -> Prop:=
  | nulP0: forall a, a = 0 -> nulP (Pc a)
  | nulPX: forall p a,
           nulP p -> a = 0 -> nulP (PX p a).
```

Une autre facon de voir est de considérer que les preuves de (nulP p) sont uniquement les arbres construits avec les deux constructeurs nulP0 et nulPX. Même définition inductive pour l'égalité, avec 4 axiomes, un pour chaque cas possible des deux arguments p et q de (eqP p q) :
(en plus ici on donne une notation == pour eqP)

```
Reserved Infix "==" (at level 70).

Inductive eqP:Pol -> Pol -> Prop :=
  | eqPc: forall a b:Z, a = b -> a == b
  | eqPX: forall p q a b,
          eqP p q -> a = b -> X*(p)+a == X*(q)+b
  | eqP0: forall p a, nulP p -> X*(p)+a == a
  | eqP0rev: forall p, forall a:Z, nulP p -> a == X*(p)+a
where "p == q" := (eqP p q).
```

2.0.5. *Preuves.* Première preuve avec Coq : l'égalité est réflexive.

```
Lemma eqP_reflexive: forall p, p == p.
Coq:
1 subgoal

  ============================
   forall p : Pol, p == p
```

Coq entre dans un processus de preuve et affiche les "buts" à démontrer. Ici il n'y en a qu'un pour l'instant. Sous la barre, c'est ce que l'on doit démontrer, au-dessus, c'est le contexte, i.e. les hypothèses locales à la démonstration. Pour l'instant il n'y en a pas. Procédons par récurrence sur p : c'est la tactique `induction` qui prend en paramètre la variable sur laquelle porte la récurrence.

```
induction p.
Coq:
2 subgoals

  z : Z
  ============================
   z == z

subgoal 2 is:
 (X  * (p)+ z) == (X  * (p)+ z)
```

deux cas se présentent, selon que p est construit avec PC ou PX ; le premier n'est pas clair à cause de la coercion $Z->Pol$: affichons cette coercion :

```
Set Printing Coercions.
Show.
Coq:
2 subgoals

  z : Z
  ============================
   Pc z == Pc z
```

```
subgoal 2 is:
 (X  * (p)+ z) == (X  * (p)+ z)
```

maintenant les coercions apparaissent ; une preuve de == se fait avec un des 4 constructeurs eqPc, eqPX, eqP0, ou eqP0rev ; manifestement, c'est eqPc qui convient, car il conclue sur ce qu'on veut montrer. on l'applique et il restera à montrer son hypothèse :

```
apply eqPc.
Coq:
2 subgoals

  z : Z
  ============================
   z = z

subgoal 2 is:
 (X  * (p)+ z) == (X  * (p)+ z)
```

qui est la réflexivité de l'égalité de Coq : la tactique dédiée "réflexivity"fait le travail :

```
reflexivity.
Coq:
1 subgoal

  p : Pol
  z : Z
  IHp : p == p
  ============================
   (X  * (p)+ z) == (X  * (p)+ z)
```

recachons les coercions

```
Unset Printing Coercions.
```

le deuxième cas de la récurrence, laissé en attente, apparaît, puisqu'on a fini le premier ; regardons dans la définition de == quel constructeur de preuve peut prouver ce but : c'est eqPX

```
apply eqPX.
Coq:
2 subgoals

  p : Pol
  z : Z
  IHp : p == p
  ============================
   p == p

subgoal 2 is:
 z = z
```

le but est exactement l'hypothèse IHp, qui l'hypothèse de récurrence.

```
exact IHp.
Coq:
1 subgoal

  p : Pol
  z : Z
  IHp : p == p
  ============================
   z = z
```

comme tout à l'heure

```
reflexivity.
```

```
Coq:
Proof completed.
```

la preuve est finie, on la sauvegarde

```
Qed.
```

la preuve est un terme dont le type est l'énoncé du lemme :

```
Print eqP_reflexive.
Coq:
eqP_reflexive =
fun p : Pol =>
Pol_ind (fun p0 : Pol => p0 == p0) (fun z : Z => eqPc z z (eq_refl z))
  (fun (p0 : Pol) (IHp : p0 == p0) (z : Z) => eqPX p0 p0 z z IHp (eq_refl z))
  p
     : forall p : Pol, p == p
```

Continuons avec une preuve très facile, utile pour la suite : montrons que Pc est compatible avec l'égalité ==

```
Lemma Pc_comp: forall a b, a = b -> Pc a == Pc b.
```

mettons les variables quantifiées dans le contexte, i.e. au-dessus de la barre

```
intros.
```

montrons les coercions :

```
Set Printing Coercions. Show.
```

puisque H dit a = b, réécrivons a en b dans le but

```
rewrite H.
```

il ne reste plus qu'à appliquer la réflexivité de ==, que l'on vient juste de démontrer :

```
apply eqP_reflexive.
Qed.
```

On continue avec une preuve moins facile

```
Lemma Pc_injective: forall a b, Pc a == Pc b -> a = b.
```

intros met aussi les prémisses d'implications dans le contexte

```
intros.
```

la forme de H indique qu'elle est forcément une preuve de == construite par le constructeur eqPc ; la tactique inversion fait se travail de déduction, en utilisant le fait qu'un prédicat défini inductivement est le plus petit prédicat vérifiant ses axiomes :

```
inversion H.
```

l'important est de voir apparaître H2, qui donne la prémisse de eqPc, parmi des hypothèse parasites

```
exact H2.
Qed.
```

recachons les coercions pour la suite

```
Unset Printing Coercions.
```

Un peu plus dur

```
Lemma eqP_symetrique: forall p q, p == q -> q == p.
```

lancons-nous dans une double récurrence, sur p et q ; le point virgule entre deux tactiques indique que la deuxième s'applique à chaque but généré par la première ; ici la première produit 2 buts, un pour chaque cas possible pour p, et la seconde un pour chaque cas de q, ce qui fait 4 buts au total :

```
induction p; induction q.
```

attaquons le premier but

```
intros.
```

pas le choix, ce but se prouve avec le premier axiome de la définition de ==, eqPc

```
apply eqPc.
```

comme au-dessus, on peut déduire z0 = z de H, car forcément H ne peut avoir été prouvée que par eqPc

```
inversion H.
```

remplacons z par z0, ce qui est justifié par H2 :

```
rewrite H2.
```

bon, Coq considère que cela est trivial

```
trivial.
intros.
```

utilisons H, qui ne peut avoir été prouvée que par eqP0rev :

```
inversion H.
```

remarquons que cette tactique a changé z en z0 dans le but

```
apply eqP0. exact H2.
```

troisième but

```
intros. inversion H. apply eqP0rev. exact H1.
```

dernier cas, le cas général de la double récurrence

```
intros. inversion H. apply eqPX.
```

appliquons l'hypothèse de récurrence :

```
apply IHp. exact H3.
```

on aurait pu réécrire avec H5, utilisons la symétrie de l'égalité de Coq

```
symmetry. exact H5.
Qed.
```

Encore un lemme utile plus tard : px est essentiellement la même chose que le constructeur PX

```
Lemma pxPX: forall p a, px p a == X*(p)+a.
induction p; intros.
```

calculons px, en le remplacant par sa définition

```
unfold px.
```

aie... c'est un peu compliqué, le résultat dépend de l'entier z ; dans Z les entiers ont trois constructeurs possibles : Z0, Zpos et Zneg, les deux derniers prenant en argument un entier strictement positif ; il faut donc, si on veut avancer, discuter selon ces 3 cas.

```
case_eq z.
```

premier cas, z = 0

```
intros. apply eqP0rev. apply nulP0. trivial.
```

deuxième cas, z est positif

```
intros. apply eqP_reflexive.
```

deuxième cas, z est négatif

```
intros. apply eqP_reflexive.
```

reste le deuxième cas de la récurrence, facile.

```
unfold px. apply eqP_reflexive.
Qed.
```

Deux polynômes nuls sont égaux

```
Lemma nulP_egal: forall p q:Pol, nulP p -> nulP q -> p == q.
```

double récurrence sur p et q, cela donne 4 cas à traiter

```
induction p; induction q; intros.
inversion H; inversion H0. rewrite H2; rewrite H4. apply eqP_reflexive.
inversion H; inversion H0.
```

remplacons z par z0, on montrera leur égalité plus tard.

```
replace z  with z0. apply eqP0rev. trivial.
  rewrite H2; rewrite H6. trivial.
inversion H; inversion H0. replace z  with z0. apply eqP0. trivial.
  rewrite H4; rewrite H6. trivial.
inversion H; inversion H0. replace z  with z0. apply eqPX.
  apply IHp. trivial. trivial.  trivial. rewrite H4; rewrite H8.
  trivial.
Qed.
```

Premier exercice sans correction : si on n'arrive pas à le faire, on peut l'admettre, pour pouvoir continuer :-)

```
Lemma nulP_comp: forall p q, nulP p -> p == q -> nulP q.
Admitted.
```

Dans la suite, pas de tactique nouvelle, chaque lemme peut être fait comme un exercice -pas facile- en utilisant les précédents

```
Lemma eqP_transitive: forall p q r, p == q -> q == r -> p == r.
induction p; induction q; induction r; intros.
inversion H; inversion H0. rewrite H3. rewrite H6. apply eqP_reflexive.
inversion H; inversion H0. rewrite H3; trivial.
inversion H; inversion H0. apply eqP_reflexive.
inversion H; inversion H0. rewrite H10. apply eqP0rev.
apply nulP_comp with q. trivial. trivial.
inversion H; inversion H0. rewrite H7. apply eqP0. trivial.
inversion H; inversion H0. apply eqPX. apply nulP_egal.
  trivial. trivial.  trivial.
inversion H; inversion H0. rewrite H6; rewrite H10. apply eqP0.
  apply nulP_comp with q. trivial. apply eqP_symetrique.  trivial.
inversion H; inversion H0. apply eqPX. apply IHp with q.
  trivial. trivial. rewrite H6; rewrite H12. trivial.
Qed.
```

Compatibilité de px par rapport à ==

```
Lemma px_comp: forall p q a b,
  a = b -> p == q -> px p a == px q b.
intros.
```

remplacons les px par des PX ; pour cela on utilise la transitivité de == en lui donnant la variable intermédiaire (q, dans son énoncé) puisqu'il ne peut l'inventer, car elle n'est pas dans la conclusion de eqP_transitive

```
apply eqP_transitive with (X*(p)+a).
apply pxPX.
apply eqP_transitive with (X*(q)+b).
apply eqPX. trivial. trivial.
apply eqP_symetrique. apply pxPX.
Qed.
```

On peut maintenant démontrer quelques propriétés de l'addition des polynômes

```
Lemma plusP_est_commutative: forall p q:Pol, p  .+  q == q  .+  p.
Set Printing Coercions.
induction p; induction q; intros.
```

ici on peut calculer les additions de ces polynômes constants ; c'est la tactique `simpl` qui effectue ce travail de calcul des fonctions récursives.

```
simpl. replace (z+z0) with (z0+z). apply eqP_reflexive.
```

pour montrer des égalités dans Z, qui est un anneau, on utilise une tactique dédiée

```
ring.
simpl. apply eqPX.  apply eqP_reflexive. ring.
simpl. apply eqPX.  apply eqP_reflexive. ring.
```

```
simpl. apply px_comp. ring. apply IHp.
Qed.
Unset Printing Coercions.

Lemma plusP0: forall p q, nulP p -> p  .+  q == q.
induction p; induction q; simpl; intros.
apply Pc_comp.  inversion H. rewrite H1. ring.
apply eqPX. apply eqP_reflexive.
inversion H. rewrite H1. ring.
inversion H. rewrite H3. simpl. apply eqP0. trivial.
apply eqP_transitive with (X*(p.+q)+(z+z0)).
apply pxPX. apply eqPX.  apply IHp.
inversion H. trivial. inversion H. rewrite H3. simpl.
trivial.
Qed.

Lemma plusP_comp1: forall q p p1,
  p == p1 -> p  .+  q == p1  .+  q.
induction q. intros p p1 H; inversion H; simpl.
rewrite H0. apply eqP_reflexive.
rewrite H1. apply eqPX.  trivial. trivial.
apply eqP0. trivial.
apply eqP0rev. trivial.
intros. inversion H. simpl.
rewrite H0. apply eqP_reflexive.
simpl.
apply px_comp. rewrite H1. trivial. apply IHq. trivial.
simpl. apply eqP_transitive with (px q (a + z)).
apply px_comp. trivial. apply plusP0. trivial.
induction q. apply pxPX. apply pxPX.
simpl. apply eqP_transitive with (px q (a + z)).
apply eqP_symetrique. apply pxPX.
apply px_comp. trivial.
apply eqP_symetrique. apply plusP0. trivial.
Qed.

Lemma plusP_comp2: forall p q q1,
  q == q1 -> p  .+  q == p  .+  q1.
intros. apply eqP_transitive with (q  .+  p).
apply plusP_est_commutative.
apply eqP_transitive with (q1  .+  p).
apply plusP_comp1. trivial. apply plusP_est_commutative.
Qed.
```

la compatibilité de l'addition

```
Lemma plusP_comp: forall p p1 q q1,
  p == p1 -> q == q1 -> p  .+  q == p1  .+  q1.
intros.  apply eqP_transitive with (p1  .+  q). apply plusP_comp1.
trivial.  apply plusP_comp2. trivial.
Qed.
```

un peu pénible...

```
Lemma plusP_est_associative: forall q p r,
                           (p  .+  q)  .+  r == p  .+  (q  .+  r).
intros. apply eqP_transitive with ((q  .+  r)  .+  p).
apply eqP_transitive with ((q  .+  p)  .+  r).
```

```
apply plusP_comp. apply plusP_est_commutative. apply  eqP_reflexive.
generalize r; generalize p; clear r; clear p.
```

je prefère cette présentation symétrique en p et r qui va donner une belle hypothèse de récurrence dans le cas numéro 9 qui est le cas général...

```
induction q; induction p; induction r; simpl; intros.
apply Pc_comp. ring.
apply eqPX. apply eqP_reflexive. ring.
apply eqPX. apply eqP_reflexive. ring.
apply px_comp. ring. apply plusP_est_commutative.
apply eqPX. apply eqP_reflexive. ring.
apply eqP_transitive with (X*(q .+ r)+ (z + z0 + z1)).
apply pxPX. apply eqP_transitive with ((X*(q .+ r)+(z + z1)).+z0).
simpl. apply eqPX. apply eqP_reflexive. ring.
apply eqP_symetrique. apply plusP_comp.
apply pxPX. apply eqP_reflexive.
apply eqP_transitive with ((X*(q .+ p)+ (z + z0)) .+ z1).
apply plusP_comp. apply pxPX. apply eqP_reflexive.
apply eqP_transitive with (X*(q .+ p)+ (z + z1 + z0)).
simpl.  apply eqPX. apply eqP_reflexive. ring.
apply eqP_transitive with (X* (q .+ p)+ (z + z1 + z0)).
 apply eqPX. apply eqP_reflexive. ring.
apply eqP_symetrique. apply pxPX.
 apply eqP_transitive with ((X*(q .+ p)+(z + z0)) .+ (X  * (r)+ z1)).
apply plusP_comp. apply pxPX. apply eqP_reflexive.
apply eqP_symetrique.
apply eqP_transitive with ((X*(q .+ r)+(z + z1)) .+ (X  * (p)+ z0)).
apply plusP_comp. apply pxPX. apply eqP_reflexive.
simpl. apply px_comp. ring. apply IHq.
apply plusP_est_commutative.
Qed.
```

Continuons avec la multiplication.

multiplication par une constante non nulle

```
Function multZP(a:Z)(p:Pol){struct p}:Pol :=
  match p with
    | Pc b => Pc (a * b)
    | PX p1 b => PX (multZP a p1) (a * b)
  end.
```

puis de deux polynômes

```
Function multP(p q:Pol){struct p}:Pol :=
  match p with
   | Pc a => multZP a q
   | PX p1 a => plusP (PX (multP p1 q) 0) (multZP a q)
  end.

Infix ".*":= multP (at level 40, left associativity).
Check (X * (3)+ 1) .* (X * (3)+ 2) .* (X * (3)+ 2).
Eval compute in (X * (3)+ 1) .* (X * (3)+ 1) .* (X * (3)+ 1).
```

la puissance

```
Fixpoint exp(p:Pol)(n:nat):Pol:=
  match n with
    | O => Pc 1
    | S n1 => p .* (exp p n1)
  end.
```

quelques tests

```
Definition p1:= X*(2)+1.
Time Eval compute in exp p1 100.
```

en compilant en byte-code, c'est plus rapide

```
Time Eval vm_compute in exp p1 100.
```

la compilation en code natif fonctionne expérimentalement, mais pas sur la version de Coq distribuée actuellement

On va maintenant s'intéresser aux polynômes normaux, i.e. dont le coefficient de plus haut degré est non nul, pour montrer que l'addition de deux polynômes normaux donne un polynôme normal.

```
Fixpoint coef1(p:Pol):Z:=
  match p with
    | Pc a => a
    | PX p1 a => coef1 p1
  end.

Inductive normal1:Pol -> Prop:=
  | normal1Pc: forall a, a <> 0 -> normal1 (Pc a)
  | normal1PX: forall p a, normal1 p -> normal1 (X*(p)+a).

Lemma normal1notnul: forall p, normal1 p -> not (nulP p).
```

Coq utilise la notation pour la négation

```
induction p; simpl;intros.
inversion H.
```

Coq, la négation not A est définie par A -> False ; déplions sa définition :

```
unfold not. intros. inversion H2. absurd (z=0);trivial.
inversion H. unfold not. intros. inversion H3. unfold not in IHp.
  apply IHp. trivial. trivial.
Qed.

Lemma normal1coefnonnul: forall p, normal1 p -> coef1 p <> 0.
induction p; simpl;intros.
inversion H. trivial.
inversion H. apply IHp. trivial.
Qed.

Definition normal(p:Pol):=
  p = Pc 0 \/ normal1 p.

Lemma normalXrev: forall p a, normal (X*(p)+a) -> normal p.
unfold normal in *; intros.
destruct H. inversion H.
right. inversion H. trivial.
Qed.

Lemma normalPc: forall a, normal (Pc a).
unfold normal in *; intros.
```

discutons selon que a est nul ou non ; on utilise Z_eq_dec, qui a pour type :

```
 forall x y : Z, {x = y} + {x <> y}
```

i.e. un ou exclusif, permettant de décider l'égalité dans Z

```
case (Z_eq_dec a 0).
intros. rewrite e. left. trivial.
intros. right. apply normal1Pc. trivial.
Qed.
```

```
Lemma normalplusP: forall p q, normal p -> normal q -> normal (p .+ q).
induction p; induction q; simpl; intros.
apply normalPc.
unfold normal in *.
destruct H0. inversion H0. right. apply normal1PX.
inversion H0. trivial.
unfold normal in *.
destruct H. inversion H. right. apply normal1PX.
inversion H. trivial.
unfold px. case_eq (p .+ q); intros.
case_eq z1;intros.
apply normalPc.
unfold normal. right. apply normal1PX. apply normal1Pc.
unfold not; intros.
```

H3 est absurde, car deux constructeurs (ici de Z) ne peuvent être égaux, son inversion conduit à supposer False et donc à tout prouver

```
inversion H3.
unfold normal. right. apply normal1PX. apply normal1Pc.
unfold not; intros. inversion H3.
rewrite <- H1.
case (IHp q); intros.
apply normalXrev with z. trivial.
apply normalXrev with z0. trivial.
rewrite H1 in H2. inversion H2.
unfold normal. right. apply normal1PX. trivial.
Qed.
```

Exercice : montrer que la multiplication de deux polynômes normaux donne un polynôme normal.

3. Preuves algébriques de théorèmes de géométrie

Cette section est directement tirée d'un article écrit avec Benjamin Grégoire et Laurent Théry [12].

De nombreux théorèmes de géométrie élémentaire (Desargues, Pappus, Feuerbach, etc) peuvent être démontrés automatiquement. Il existe différentes méthodes efficaces, dont la plus connue est dans doute celle de Wu, mais qui se prêtent plus ou moins à la formalisation.

Nous allons étudier une méthode classique qui consiste à traduire les propriétés géométriques en équations polynomiales, puis à réduire le théorème de géométrie à un problème d'appartenance à un idéal. Ce qui est moins classique, c'est de produire une preuve formelle acceptable par un assistant de preuve.

La méthode consiste à utiliser un calcul incomplet de bases de Groebner pour produire un certificat de petite taille (un programme sans boucle, ou straightline program), permettant de démontrer ensuite une égalité polynomiale de type Nullstellensatz : si $P_1 ... P_n$ sont des polynômes exprimant les hypothèses du théorème, et P sa conclusion, alors on peut écrire $cP^r = Q_1P_1 + ... + Q_nP_n$. Le calcul du certificat représentant $(c, r, Q_1, ..., Q_n)$ peut être fait dans n'importe langage (par exemple ocaml ou F7), alors que la preuve du Nullstellensatz est produite par réflexion dans le système d'assitance à la preuve Coq. On montrera sur des exemples dans le système Coq les différentes étapes et principes employés.

L'intégration sûre de procédures de décisions dans les assistants de preuve (comme Coq) est actuellement un sujet très actif actuellement. Les deux mondes du calcul formel et de la preuve sont en train de se rejoindre, avec des expériences comme celles de Thomas Hales (conjecture de Kepler) et Georges Gonthier (classification des groupes finis). Enfin, disons plutôt que pour l'instant, c'est le monde de la preuve qui fait de plus en plus de calcul formel, ou au moins qui utilise des algorithmes mathématiques.

Il existe de nombreuses procédures de décision qui sont bien connues mais il est difficile de les rendre disponibles dans un assitant de preuve : on doit apporter la preuve formelle que leurs résultats sont corrects. Cela peut expliquer pourquoi très peu de théorèmes de géométrie élémentaire ont été prouvés formellement (voir la liste compilée par Freek Wiedijk [26]).

L'intégration d'une procédure de décision dans un assistant de preuve peut se faire de plusieurs manières. On peut tout faire dans l'assistant : on écrit la procédure comme une tactique (une tactique est une procédure qui permet de construire un pas de preuve), et, à chaque fois qu'on l'utilise, une preuve est générée pour le cas particulier résolu. Ou bien, si le système offre un langage de programmation, on peut programmer la procédure de décision, et prouver sa correction en tant qu'algorithme une fois pour toutes. Dans ces deux cas, cela demande beaucoup d'efforts et de temps, surtout si la procédure est complexe.

Une troisième voie consiste à utiliser des programmes externes à l'assistant de preuve. Par exemple, on peut prendre une implémentation efficace de la procédure, et la modifier (!) pour qu'elle produise une trace de ses calculs. Ensuite, l'assistant n'a plus qu'à suivre cette trace pour construire sa propre preuve. Une autre possibilité est d'utiliser un certificat à la place d'une trace. Un certificat ne contient pas toute l'éxécution des calculs, mais juste assez d'information pour vérifier que le résultat est correct et en fournir une preuve facilement. On peut trouver des exemples de cette méthode pour certifier la primilaté de nombres [13] ou le fait qu'un polynôme est positif [17].

Vérifier des certificats est souvent plus difficile que suivre des traces mais bien plus facile que de programmer la procédure dans l'assistant ; l'avantage est qu'un certificat est bien plus petit qu'une trace. On peut vérifier un certificat soit avec une tactique, soit avec un programme vérifié.

Dans ce cours on va étudier un cas où on utilise un programme externe pour produire un certificat, et un programme interne pour vérifier ce certificat et produire une preuve. Il s'agira de prouver des formules du type suivant :

$$\begin{array}{l}\forall X_1, \ldots, X_n \in R,\\ P_1(X_1, \ldots, X_n) = 0 \wedge \ldots \wedge P_s(X_1, \ldots, X_n) = 0\\ \Rightarrow P(X_1, \ldots, X_n) = 0\end{array}$$

où R est un anneau commutatif intègre et $P, P_1, \ldots, P_s$ sont des polynômes.

Ce problème a été intensivement étudié dans les années 80, en particulier par la communauté du calcul formel, et la principale procédure utilisée est l'algorithme de Buchberger [3]. Beaucoup d'efforts ont conduit à des variantes de plus en plus efficaces, et le domaine où cela a été le plus efficace est celui de la démonstration automatique de théorèmes de géométrie (consulter par exemple [6, 19, 20, 24, 27, 21] pour un tour d'horizon). C'est pourquoi on choisira nos exemples dans la géométrie.

Dans le contexte des assistants de preuve, les choses sont plus récentes. John Harrison [16] utilise une implémentation basique de l'algorithme de Buchberger pour produire un certificat d'arithmétique élémentaire (par exemple pour prouver le théorème des restes chinois) dans le système [14]. Amine Chaieb and Makarius Wenzel [4] utilisent l'algorithme de Buchberger avec le système Isabelle system [22]. J'avais fait la même chose avec Coq et F7 (voir [23]), qui est une des implémentations les plus efficaces pour calculer des bases de Groebner.

On va prendre comme exemples des théorèmes déjà prouvés avec la méthode de Wu, comme les thèorèmes de Desargues, Pascal, et une vingtaine d'autres. Ils sont assez simples, mais obtenir un certificat utilisable pour construire une preuve formelle de ces théorèmes n'est pas aussi simple que de les vérifier par le calcul, ce qui n'est déjà pas forcément un problème trivial.

3.1. **Géométrie et polynômes.** On se réduit aux prédicats géométriques exprimables par la nullité d'un polynôme en les coordonnées des points. Disons qu'on se place dans R^n, où R est un anneau intègre. Par exemple, on définit en Coq les points du plan :

```
Record point:Type:={
 X:R;
 Y:R}.
```

puis la colinéarité de 3 points :

```
Definition collinear(A B C:point):=
  (X A - X B)*(Y C - Y B)-(Y A - Y B)*(X C - X B)=0.
```

le fait que (A B) est parallèle à (C D) :

```
Definition parallel (A B C D:point):=
  ((X A)-(X B))*((Y C)-(Y D))=((Y A)-(Y B))*((X C)-(X D)).
```

et sa négation (on utilise une variable auxiliaire x) :

```
Definition notparallel (A B C D:point)(x:R):=
  x*(((X A)-(X B))*((Y C)-(Y D))-((Y A)-(Y B))*((X C)-(X D)))=1.
```

Prenons l'exemple du théorème de Desargues :

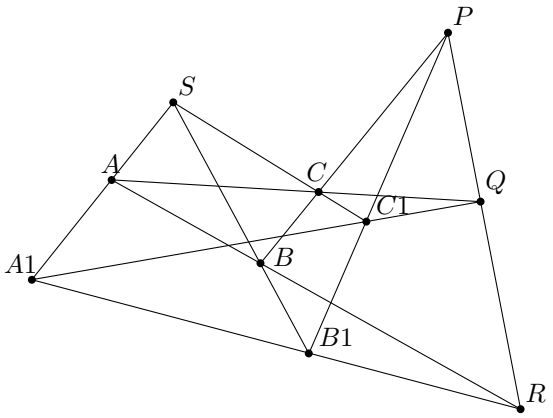

Figure 3.1: Théorème de Desargues

Pour commencer la preuve, on écrira en Coq :

```
Lemma Desargues: forall A B C A1 B1 C1 P Q R S:point,
  X S = 0 -> Y S = 0 -> Y A = 0 ->
  collinear A S A1 -> collinear B S B1 -> collinear C S C1 ->
  collinear B1 C1 P -> collinear B C P ->
  collinear A1 C1 Q -> collinear A C Q ->
  collinear A1 B1 R -> collinear A B R ->
  collinear P Q R
  \/ X A = X B \/ X A = X C \/ X B = X C  \/ X A = 0 \/ Y B = 0 \/ Y C = 0
  \/ collinear S B C \/ parallel A C A1 C1 \/ parallel A B A1 B1.
```

Le symbole
`/` désigne la disjonction logique. On voit que le théorème conclue sur `collinear P Q R`, mais ce n'est pas vrai tout le temps, il y a des cas dégénérés, qui sont justement donnés par les disjonctions. On a aussi fixé les coordonnées d'un point, et l'abcisse d'un autre pour accélérer les calculs,sans perdre de généralité.

En dépliant les définitions des prédicats géométriques, en rempla cant les disjonction par des produits de polynômes on obtient le "but" suivant [1] :

```
  A : point
  B : point
  C : point
  A1 : point
  B1 : point
  C1 : point
  P : point
  Q : point
  R : point
```

1. un "but" en Coq est une formule du type $f_1, ..., f_n \Rightarrow f$, où $f_1, ..., f_n$ sont les hypothèses -ou "contexte"- et f est la formule à prouver ; les hypothèses sont placées au dessus de la barre, et la formule à prouver en dessous

```
S : point
H2 : (X A - 0) * (Y A1 - 0) - (0 - 0) * (X A1 - 0) = 0
H3 : (X B - 0) * (Y B1 - 0) - (Y B - 0) * (X B1 - 0) = 0
H4 : (X C - 0) * (Y C1 - 0) - (Y C - 0) * (X C1 - 0) = 0
H5 : (X B1 - X C1) * (Y P - Y C1) - (Y B1 - Y C1) * (X P - X C1) = 0
H6 : (X B - X C) * (Y P - Y C) - (Y B - Y C) * (X P - X C) = 0
H7 : (X A1 - X C1) * (Y Q - Y C1) - (Y A1 - Y C1) * (X Q - X C1) = 0
H8 : (X A - X C) * (Y Q - Y C) - (0 - Y C) * (X Q - X C) = 0
H9 : (X A1 - X B1) * (Y R - Y B1) - (Y A1 - Y B1) * (X R - X B1) = 0
H10 : (X A - X B) * (Y R - Y B) - (0 - Y B) * (X R - X B) = 0
============================
 ((X P - X Q) * (Y R - Y Q) - (Y P - Y Q) * (X R - X Q) - 0) *
 ((X A - X B) *
  ((X A - X C) *
   ((X B - X C) *
    ((X A - 0) *
     ((Y B - 0) *
      ((Y C - 0) *
       (((0 - X B) * (Y C - Y B) - (0 - Y B) * (X C - X B) - 0) *
        (((X A - X C) * (Y A1 - Y C1) - (0 - Y C) * (X A1 - X C1)) *
         ((X A - X B) * (Y A1 - Y B1) - (0 - Y B) * (X A1 - X B1))))))))))) =
0
```

Le problème est donc réduit maintenant à prouver une formule du type $P_1 = 0, \ldots, P_s = 0 \Rightarrow P = 0$

3.2. $P_1 = 0, \ldots, P_s = 0 \Rightarrow P = 0$ **: Nullstellensatz et bases de Groebner.** On cherche à prouver une formule du type :

$$\begin{array}{l}\forall X_1, \ldots, X_n \in R, \\ P_1(X_1, \ldots, X_n) = 0 \wedge \ldots \wedge P_s(X_1, \ldots, X_n) = 0 \\ \Rightarrow P(X_1, \ldots, X_n) = 0\end{array}$$

Le théorème des zéros de Hilbert montre comment réduire des preuves d'égalités de polynômes à des calculs algébriques (voir par exemple [7]) : *dans un corps algébriquement clos, un polynôme P s'annule sur tous les zéros communs aux polynômes $P_1, ...P_s$, si et seulement si il existe une puissance de P qui appartient à l'idéal engendré par $P_1, ...P_s$.*

Il est facile de voir que si un polynôme P dans $R[X_1, \ldots, X_n]$ verifie $cP^r = \sum_{i=1}^{s} Q_i P_i$, avec $c \in R$, $c \neq 0$, r un entier non nul, et les Q_is dans $R[X_1, \ldots, X_n]$, alors P est nul dès que les polynômes $P_1, ..., P_s$ le sont. L'inverse est aussi vrai quand R est algébriquement clos : la méthode est complète. Ainsi, notre problème initial revient à trouver $Q_1, \ldots, Q_s$, c et r tels que $cP^r = \sum_i Q_i P_i$, i.e. à montrer que P est dans la radical de l'idéal engendré par $P_1, ...P_s$. Dans ce cas, on appelle $(c, r, Q_1, \ldots, Q_s)$ le "certificat" de la formule qu'on veut prouver, puisqu'il est facile d'obtenir une preuve à partir de ce certificat.

En pratique, on rencontrera essentiellement le cas $r-1$, les autres cas pouvant soit se traiter par énumération, soit en ajoutant une variable supplémentaire, comme expliqué par exemple dans [23].

Du coup, on se ramène à montrer qu'un polynôme est dans un idéal, ce qui se fait en utilisant des bases de Groebner. On rappelle maintenant les bases de cette technique.

3.2.1. *Division de polynômes et bases de Groebner.* Une *base de Groebner* d'un idéal et une famille de polynômes de l'idéal telle que leur monômes de tête (pour un ordre bien fondé bien choisi sur les monômes, par exemple l'ordre lexicographique ou l'ordre du degré) engendre l'idéal des monômes de tête de tous les polynômes de l'idéal. La principale propriété d'une base de Groebner est de fournir un test d'appartenance à l'idéal : un polynôme appartient à l'idéal si et seulement si sa *division* euclidienne par les polynômes de cette base donne un reste nul.

A plusieurs variables, la division euclidienne est une généralisation de la division euclidienne à une variable : pour diviser un polynôme P par un polynôme $aX^\alpha - Q$ on écrit $P = aX^\alpha S + T$ où T n'a pas de monôme multiple de X^α. Alors on change P en $QS + T$ et on répète le procédé. Quand on obtient un polynôme sans multiple de X^α, on a obtenu le reste de la division.

Par exemple, supposons que l'on utilise l'ordre du degré sur les monômes. Le monôme de tête de $x^2 - z$ est x^2. Pour diviser $x^4y + x^2 - 1$ par $x^2 - z$, on réécrit x^2 en z partout. Cela donne $z^2y + z - 1$, pas divisible par $x^2 - z$, c'est donc le reste de cette division. Pour diviser un polynôme par une famille de polynômes, on répète ce procédé avec les polynômes de la famille, tant que c'est possible.

3.2.2. *Unicité du reste.* En général, le reste de la division par une famille n'est pas unique : il dépend de l'ordre dans lequel on a choisi les polynômes dans la famille. Avec une base de Groebner, ce reste est unique : c'est une caractéristique des bases de Groebner.

Par exemple, la division de $x^2y^2 - y^4$ par $\{x^2 + 1, xy - 1\}$ donne le reste $-y^2 - y^4$ si on divise par $x^2 + 1$ mais donne $1 - y^4$ si on divise par $xy - 1$. Les deux restes sont irréductibles : alors la famille n'est pas une base de Groebner.

On peut montrer que la famille complétée $\{x^2+1, xy-1, x+y, y^2+1\}$ est une base de Groebner de l'idéal engendré par $\{x^2 + 1, xy - 1\}$. Chaque division de $x^2y^2 - y^4$ par cette famille donne le reste 0. Dans ce cas, avec simplement un algorithme de division d polynômes, on peut conclure que le polynôme $x^2y^2 - y^4$ est dans l'idéal engendré par $\{x^2 + 1, xy - 1\}$.

3.2.3. *Calcul paresseux.* Considérons maintenant $x^3 - y$, qui a un degré en y strictement plus petit que 2. Clairement, le diviser par les 3 premiers polynômes de la base $\{x^2 + 1, xy - 1, x + y\}$ est suffisant pour obtenir le reste 0. Cela montre que dans certains cas, il n'est pas nécessaire d'avoir une base de Groebner complète, pour conclure. Quand on sait que, dans un calcul de base de Groebner avec l'algorithme de Buchberger, la moitié finale du temps est utilisée pour vérifier que la famille est bien une base de Groebner sans produire de polynômes nouveaux, on comprend qu'il peut être utile d'utiliser un calcul "paresseux" de bases de Groener : à chaque nouveau polynôme ajouté dans la base de Groebner de $P_1, ..., P_s$ au cours de son calcul, on teste si P se réduit à 0 par division avec la nouvelle base. En pratique c'est très efficace.

3.3. **Algorithme de Buchberger et certificats.** La principale méthode pour calculer une base de Groebner est l'algorithme de Buchberger. Il consiste à compléter la famille initiale par de nouveaux polynômes appelés *S-polynomes*. Pour une paire de polynômes (P, Q), son S-polynome est $t_1P - t_2Q$ où t_1 and t_2 sont des monômes choisis de manière à rendre les monômes de tête de t_1P et t_2Q identiques pour qu'ils s'annulent dans la soustraction. L'algorithme commence avec la famille initiale et y ajoute tous les restes non nuls des divisions des S-polynômes de la famille. Puis recommence, jusqu'à ce que les restes soient tous nuls. Le lemme de Dixon assure que cet algorithme termine. La famille ainsi complétée est une base de Groebner. Cet algorithme a été pour la première fois prouvé formellement par [25], en Coq.

Pour tester l'appartenance de P à l'idéal engendré $P_1, ...P_s$, on peut le modifier simplement : chaque fois qu'un reste non nul de S-polynôme est ajouté, on divise P par ce reste et on remplace P par le reste de cette division. S'il est nul, on a gagné, P est dans l'idéal, et on arrête. Pour récupérer les polynômes $Q_1, ..., Q_s$ tels que $cP = \sum_i Q_iP_i$, il suffit de se souveneir des divisions effectuées au cours de l'algorithme.

3.3.1. *Exemple.* On veut montrer que $x^2 + 1 = 0 \wedge xy - 1 = 0 \Rightarrow x^3 - y = 0$. Soient $P_1 = x^2 + 1$, $P_2 = xy - 1$ et $P = x^3 - y$. La famille initiale est $\{P_1, P_2\}$ et on cherche un certificat c, Q_1, Q_2 tel que $cP = Q_1P_1 + Q_2P_2$. On commence par diviser P par $\{P_1, P_2\}$ de gauche à droite. P_1 divise P et le reste est $R_1 = -x - y = P - xP_1$. R_1 est non nul et irréductible par $\{P_1, P_2\}$, on commence donc la complétion. Il y a un seul S-polynome pour $\{P_1, P_2\}$, qui est $P_3 = x + y = yP_1 - xP_2$. Il est irréductible par $\{P_1, P_2\}$, on l'ajoute donc à $\{P_1, P_2\}$. On essaye de diviser R_1 par $\{P_1, P_2, P_3\}$, ce qui donne $0 = R_1 + P_3$ et on peut donc arrêter la complétion. On a

$$0 = R_1 + P_3 = (P - xP_1) + (yP_1 - xP_2)$$

en sortant P on a

$$P = (x - y)P_1 + xP_2$$

et le certificat $c = 1$, $Q_1 = x - y$ and $Q_2 = x$. Notons que dans ce cas on obtenu le certificat sans calculer toute la base de Goebner.

3.3.2. *Programme sans boucle - Straight-line programs.* L'exemple précédent est simple, mais en pratique, les Q_i peuvent être vraiment très gros. Pour éviter cela on va utliser une forme de certificat plus efficace, en utilisant la trace des calculs conduisant aux Q_i. Elle est composée de deux parties.

La première, *CR*, est une liste de polynômes. Elle donne les coefficients permettant d'exprimer P comme combinaison linéaire des polynômes de départ $P_1, ..., P_s$ et des polynômes $P_{s+1}, ..., P_{s+p}$que le calcul de base de Groebner a ajoutés jusqu'à ce que P soit réduit à 0.

La deuxième, C, est une liste de liste de polynômes, une pour chaque P_{s+i}, et qui contient les coefficients qui expriment P_{s+i} comme combinaison linéaire des $P_1, ..., P_{s+i-1}$.

Plus précisément :

$$\begin{array}{rl} CR = & [c_1, \ldots, c_{s+p}] \\ C = & [[a_{1\ s+1}, \ldots, a_{s\ s+1}], \\ & \ldots \\ & [a_{1\ s+p}, \ldots, a_{s\ s+p}, \ldots, a_{s+p-1\ s+p}]] \end{array}$$

avec

$$\forall i \in [1; p], P_{s+i} = a_{1\ s+i}P_1 + \ldots + a_{s+i-1\ s+i}P_{s+i-1}$$

et

$$P = -(c_1P_1 + \ldots + c_{s+p}P_{s+p})$$

Pour simplifier, on a supposé ici que R est un corps (donc $c = 1$). Mais il est facile d'étendre tout cela au cas d'un anneau intègre, où on effectue des pseudo-divisions. Comme chaque polynôme est défini à l'aide des précédents, C a une structure de programme sans boucle (straight-line program, ou SLP). Dans notre exemple, cela donne :

$$\begin{array}{l} P_1 := x^2 + 1; \\ P_2 := xy - 1; \\ P_3 := yP_1 + (-x)P_2; \\ P := -((-x)P_1 + 0P_2 + 1P_3); \end{array}$$

ainsi le certificat est $CR = [-x, 0, 1]$ et $C = [[y, -x]]$.

L'avantage principal des SLP est qu'ils permettent de partager les calculs. Cela peut changer un coût exponentiel en un coût linéaire (par exemple [8] où l'utilisation de SLP donne une borne de complexité).

En général, un SLP est une suite d'affectations de variables, chacune dépendant des précédentes :

$$\begin{array}{l} x_1 := f_1(); \\ x_2 := f_2(x_1); \\ x_3 := f_3(x_1, x_2); \\ ... \\ x_n := f_n(x_1, ..., x_{n-1}); \end{array}$$

où $f_1, f_2, ..., f_n$ sont des procédures. En calcul formel, ce sont des fractions rationnelles, ici on se limite aux polynômes. Un SLP peut être aussi vu comme un graphe sans cycle, i.e. un arbre qui partage ses sous-arbres.

Voici un exemple dans notre contexte où la complexité chute avec les SLP :

Soit f_n le $n$ième nombre de Fibonacci : $f_0 = 0, f_1 = 1, f_{n+2} = f_{n+1} + f_n$ et supposons qu'on veuille montrer que $X^{f_{n+1}} - 1 = 0 \wedge X^{f_n} - 1 = 0 \Rightarrow X - 1 = 0$. Le calcul d'une base de Groebner de $\{X^{f_{n+1}} - 1, X^{f_n} - 1\}$ suit l'algorithme d'Euclide de calcul du pgcd, et on obtient :

$$X - 1 = P_{n-2}(X^{f_{n+1}} - 1) + P_{n-1}(X^{f_n} - 1) \tag{3.1}$$

où le polynôme P_n est défini par $P_0 = 0, P_1 = 1, P_n = -X^{f_n}P_{n-1} + P_{n-2}$. Le certificat en SLP est :

$$\begin{array}{rl} CR = & [0, \ldots, 0, 1] \\ C = & [[1, -X^{f_{n-1}}], \\ & [0, 1, -X^{f_{n-2}}], \\ & \ldots \\ & [0, \ldots, 0, 1, -X^{f_2}]] \end{array} \tag{3.2}$$

```
inideal(P,F){
   (* F = [P1,...,Pn] *)
   R := P; C := [];CR := LR; R := R1;
   SP:= Spolynomials(F,F);
   let (R1,LR) = divide(R,F) in
   while R <> 0 do (* stop if P divides to 0 by F *)
     if SP = [] then Fail
        (* the Groebner basis is computed without reducing P to 0 *)
     else let (S,LS)::SP1 = SP in
          SP := SP1;
          let (D,LD) = divide(S,F) in
          if D <> 0 (* add a new polynomial to F *)
            then F := F + [D];   (* + denotes concatenation of lists *)
                 SP := SP + Spolynomials(F,[D]);
                 C := C + [merge(LD,LS)];
                 let (R1,LR) = divide(R,F) in (* reduce R by F *)
                 CR := merge(CR,LR);
                 R := R1;
   done;
   return(CR,C);
}
```

Figure 3.2: Pseudo-code de l'agorithme de Buchberger tronqué avec certificat

Supposons que pour tester une égalité entre polynômes, on applique d'abord la distributivité et ensuite on regroupe les monômes égaux. Comparons la vérification des deux certificats (3.1) et (3.2) en termes d'opérations sur les monômes. Dans le premier, P_n a le degré $f_{n+2} - 2$ et a f_n monômes, avec le coefficient 1 (n impair) ou -1 (n pair), et la vérification de cette équation demande $2f_n$ multiplications, et $3f_n$ additions. Dans le second, chaque polynôme intermédiaire a la forme $X^{f_k} - 1$, et la vérification ne demande que $2n - 4$ multiplications et $4n - 8$ additions. Comme $f_n \sim ((1 + \sqrt{5})/2)^n/\sqrt{5}$, il y a un facteur exponentiel entre les deux vérifications.

3.3.3. *Pseudo-code.* Pour terminer cette subsection, voilà le pseudo-code (Figure 3.2) de la fonction `inideal` qui génère calcule une base de Groebner partielle et le certificat si le polynôme P est dans l'idéal engendré par la famille F :

La fonction `divide` calcule le reste et la liste des quotients d'une division. La fonction `Spolynomials` calcule les S-polynômes de deux familles. Pour chaque S-polynôme, elle rend aussi les monômes et les polynômes utilisés pour le calculer. La fonction `merge` ajoute les termes de deux listes de même rang, en complétant par des zéros si besoin.

3.4. **Vérification de grands certicats en** Coq **.** Pour vérifier un cerficat $cP^r = \sum_{i=1}^{s} Q_i P_i$, sous forme de SLP ou non, les axiomes d'anneaux suffisent. On développe les produits de sommes, on regroupe les monômes égaux, on supprime ceux dont le coefficient est nul, et on les ordonne. L'égalité doit alors être triviale. Malheureusement, cette méthode génère des preuves vite gigantesques : chaque utilisation d'un axiome pour réécrire une expression (commutativité, associativité, distributivite, etc) apparaît dans la preuve.

Huereusement, le système Coq possède un langage de programmation sur lequel on peut raisonner. Cela autorise l'écriture de procédures de décision certifiées et l'utilisation de la méthode de la *réflexion.*

3.4.1. *Réflexion.* La technique de la *réflexion* introduite par Allen et al. [1] profite du mécanisme de calcul d'un système (quand il en a un) pour réduire considérablement la taille des preuves. D'une certaine manière, cette technique transforme de l'espace mémoire en temps de calcul.

Elle est fondée sur la remarque suivante :

- Soit $P : A \to \mathsf{Prop}$ un prédicat sur un ensemble A.
- Supposons que l'on est capable d'écrire dans le système un programme c tel que les propriétés suivantes soient vérifiées :

$$\mathsf{c\_spec}: \ \forall a,\ c\ a = \mathsf{true} \to P\ a$$

Autrement dit, pour toute valeur a, si l'évaluation du programme c sur a rend true alors P est satifait pour a. Cela signifie que c est une procédure de semi-décision pour la propriété P et $\mathsf{c\_spec}$ est le lemme qui exprime que cette procédure de semi-décision est correcte.

Maintenant, supposons qu'on doive prouver $P\ a'$ pour un a' donné et que, pour ce a', le système est capable de calculer $c\ a'$ en true. Pour prouver $P\ a'$, on peut appliquer le lemme $\mathsf{c\_spec}$ avec a', et il nous reste à prouver $c\ a' = \mathsf{true}$. Puisque $c\ a'$ se calcule en true, cette proposition $c\ a' = \mathsf{true}$ est identique (on dit *convertible* en Coq) pour le prouveur à la proposition $\mathsf{true} = \mathsf{true}$, qui est prouvée par la réflexivité de l'égalité.

Le système Coq est fondé sur l'isomorphisme de Curry-Howard. Cela veut dire que les preuves sont représentées par des programmes, les propositions par des types, et les preuves correctes sont les programmes bien typés. Par exemple, notre preuve de $P\ a'$ est le programme
$\mathsf{c\_spec}\ a'\ (\mathsf{refleq\ true})$.

La dérivation de typage de cette preuve est :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathsf{c\_spec}\ a' : c\ a' = \mathsf{true} \to P\ a'} \qquad \dfrac{\Gamma \vdash \mathsf{refleq\ true} : \mathsf{true} = \mathsf{true} \quad \mathsf{true} = \mathsf{true} \equiv c\ a' = \mathsf{true}}{\Gamma \vdash \mathsf{refleq\ true} : c\ a' = \mathsf{true}}[\textsc{Conv}]}{\Gamma \vdash \mathsf{c\_spec}\ a'\ (\mathsf{refleq\ true}) : P\ a'}$$

Ici le point crucial est l'utilisation de la règle de typage Conv :

$$\dfrac{\Gamma \vdash t : T \qquad T \equiv U}{\Gamma \vdash t : U}[\textsc{Conv}]$$

qui permet de voir un programme t de type T comme un programme de type U si T et U convertibles, i.e. égaux modulo calcul. Toutes les étapes de réduction nécessaires pour vérifier la convertibilité de $\mathsf{true} = \mathsf{true}$ et $c\ a' = \mathsf{true}$ n'apparaissent pas dans le terme de preuve.

Evidemment, si les étapes de réduction n'apparaissent pas dans la preuve, elles seront effectuées dans l'étape de vérification de cette preuve. Ainsi le temps de calcul nécessaire pour vérifier une preuve qui utilise la réflexion va non seulement dépendre de l'efficacité du mécanisme de calcul du système, mais aussi de la procédure de semi-décision programmée dans le système.

Le mécanisme de réduction compilé de Coq possède une stratégie très efficace pour réduire les programmes [10]. C'est pourquoi la réflexion est une technique très efficace en Coq.

3.4.2. *Réflexion et certificats.* La réflexion en Coq a été introduite par Samuel Boutin [2] pour prouver les égalités dans les anneaux. Elle a été ensuite améliorée et maintenant existe une bibliothèque efficace de calcul arithmétique sur les polynômes [11], dans laquelle sont définis deux types de données :

- $\mathcal{E}$ represente le type des expressions polynômiales, i.e. l'algèbre libre.
- $\mathcal{P}$ represente le type des polynômes (à plusieurs variables) en forme de normale de Horner.

Les opérations de base comme l'addition et la multiplication sont définies sur le type $\mathcal{P}$, il est donc facile de définir un algorithme de normalisation `norm` de $\mathcal{E}$ dans $\mathcal{P}$ par récurrence structurelle (i.e. sur la structure syntaxique des objets).

La correction des opérations de base est obtenue en utilisant une fonction d'interprétation $[\![\ ]\!]_\rho$ de $\mathcal{P}$ vers un anneau quelconque R, où ρ est la fonction d'évaluation des variables : elle donne une valeur de R à chaque indéterminée des polynômes. On a prouvé que chaque opération arithmétique est correcte par rapport à la fonction d'interprétation. Par exemple la spécification de l'addition est :

$$\forall \rho\ P_1\ P_2,\ [\![P_1 +_{\mathcal{P}} P_2]\!]_\rho = [\![P_1]\!]_\rho +_R [\![P_2]\!]_\rho$$

De même, la correction de la normalisation est définie en utilisant une fonction d'interprétation $[\![\ ]\!]^{\mathcal{E}}_\rho$ sur les expressions polynomiales :

$$\forall \rho\ E,\ [\![E]\!]^{\mathcal{E}}_\rho = [\![\mathsf{norm}\ E]\!]_\rho$$

Ainsi, pour prouver que deux expressions r_1 et r_2 de R sont égales, il suffit de trouver deux expressions polynomiales E_1, E_2 et une fonction d'évaluation ρ tels que $[\![E_i]\!]^{\mathcal{E}}_\rho$ se réduise à r_i. Si la normalisation de E_1 et E_2 donne la même forme normale de Horner, alors r_1 et r_2 sont égales.

Cette stratégie n'est pas forcément la meilleure. La normalisation est définie par une récurrence structurelle naïve :

pour normaliser $(X+Y)^{100}-(X+Y)^{100}$ la fonction normalise deux fois le sous-terme $(X+Y)^{100}$ et ensuite calcule la soustraction. ce qui n'est pas optimal...

3.4.3. *Implémentation du vérificateur de certificat.* On définit d'abord une fonction `mult_l` qui normalise chaque ligne du certificat. En Coq cela donne

```
Function mult_l (L_e L : list P) : P :=
  match L_e, L with
  | e : :L'_e, p : :L' ⇒ e *_P p +_P mult_l L'_e L'
  | _, _ ⇒ 0_P
  end.
```

Ensuite une seconde fonction `compute_list` qui regroupe tous les polynômes normalisés qui correspondent aux lignes du certificat

```
Function compute_list (LL_e : list (list P)) (L :list P) : list P :=
  match LL_e with
  | L_e : :LL_e ⇒ compute_list LL_e ((mult_l L_e L) : :L)
  | _ ⇒ L
  end.
```

Finallement la fonction de vérification `check` vérifie l'égalité des deux formes normales :

```
Function check (L_e :list E) (p :E) (certif : list (list P) * list P) :=
  let (LL_e, L'_e) := certif in
  let L := map norm L_e in
  norm p =?=_P mult_l L'_e (compute_list LL_e L).
```

Notons que toutes ces fonctions sont tail-récursives. Pour prouver la correction de ce vérificateur, on définit d'abord la propriété d'une liste d'être composée seulement de polynômes nuls :

```
Definition Allzero ρ (L : list P) := ∀P ∈ L, [[P]]_ρ = 0.
Definition Allzero_E ρ (L_e : list E) := ∀P ∈ L_e, [[P]]^E_ρ = 0.
```

On montre ensuite que les deux fonctions `mult_l` et `compute_list` se comportent bien avec les listes de polynômes nuls :

```
Lemma mult_l_spec : ∀ρ L_e L, Allzero ρ L → [[mult_l L_e L]]_ρ = 0.
Lemma compute_list_spec :
  ∀ρ LL_e L, Allzero ρ L → Allzero ρ (compute_list LL_e L).
```

Finally, we can derive the correctness of our checker

```
Lemma check_correct :
  ∀ρ L p certif, check L p certif = true → Allzero_E ρ L → [[p]]^E_ρ = 0.
```

Maintenant, définir le vérificateur et prouver sa correction est facile.

3.5. **Application à la géométrie.** Dans son livre [6], Shang-Ching Chou démontre mécaniquement 512 théorèmes de géométrie plane. On peut en montrer parmi les plus difficiles à l'aide de la méthode précédente. Et comparer les résultats à d'autres systèmes : HOL Light [14] et Macaulay2 [9]).

Comme indiqué précédemment, les points sont représentés par leurs coordonnées cartésiennes, et les prédicats géométriques par des polynômes, issus de déterminants, produits scalaires et relations algébriques entre fonctions trigonométriques.

La figure 3.3 donne une synthèse des résultats.

La machine utilisée est un PC linux à deux processeurs Intel Xeon 3.2Ghz et 33Gb de mémoire.

Les colonnes contiennent :

(1) Le nom du théorème et entre parenthèses la page dans le livre de Chou où il est énoncé (quand elle existe).

(2) Le temps en secondes pour calculer le certificat.

	Time (seconds)		Size (characters)		Size (nodes)	
Theorem	Computing	Verifying	Polynomials	Certificate	Term	SLP
Ceva (264)	181	2.5	538644	477414	266233	76669
Desargues (*)(269)	0.3	0.01	6359	4551	24527	4311
Feuerbach (199)	0.8	0.4	52569	16999	15585	5497
Pappus (*)(100)	1.3	0.2	2721	1934	29945	8031
Pascal_circle (*)	397	12	732982	864509	754290	183505
Pascal_circle2 (20)	91	2.6	10603	15128	312626	66154
Ptolemy (*)	1.1	0.5	1549	1556	26210	9129
Ptolemy_theo95 (142)	200	2.4	571931	571931	344278	73257
Pythagora	0.000	0.009	7	7	4	4
Simson (240)	0.3	0.2	1541	1238	15680	4919
Thales	0.03	0.1	5422	5169	3146	1323
bisectors	0.002	0.04	165	165	105	69
butterfly (119)	0.1	0.2	12116	11125	13661	3980
Euler circle	0.06	0.5	5532	2936	2795	1146
chords	0.002	0.04	639	642	568	282
altitudes	1.1	0.3	4947	5386	5295	1801
isosceles	0.001	0.01	10	10	3	3
medians	0.005	0.06	2910	2717	2284	1064
bisections	0.005	0.06	2577	2145	1911	831
Minh	0.07	0.1	3367	3616	3987	1881
SegmentsofChords	0.1	0.09	10375	9839	7476	2491
threepoints	0.11	0.13	2890	2587	2796	1105
fib(16)	0.003	0.8	15786	393	416	137
fib(17)	0.004	1.3	26059	423	465	151
fib(18)	0.004	2.4	40864	464	517	166
fib(22)	0.008	68	307720	630	753	225

Figure 3.3: Temps et tailles de quelques théorèmes

(3) Le temps en secondes pour vérifier le certificat.

(4) Le nombre de caractères pour écrire $c$et les Q_i lorsqu'on développe le certificat en $cP = \sum_i Q_i P_i$.

(5) Le nombre de caractères pour écrire le certificat.

(6) La taille du certificat écrit comme un terme de preuve (en nombre de neuds).

(7) La taille du certificat écrit comme un SLP optimisé : chaque sous-terme du terme de preuve est partagé (avec un opérateur *let*).

Une présentation plus détaillée des exemples est disponible sur cette page web :

`http://www-sop.inria.fr/marelle/CertiGeo`

Quelques commentaires sur ces résultats :

- Le nombre de variables des calculs de bases de Groebner est d'environ 20 (le nombre de coordonnées des points) et le degré des polynômes en entrée est en général 2 (ce qui est le cas général, puisqu'on peut transformer un polynôme de degré quelconque en un polynôme de degré au plus 2 en ajoutant des variables).
- Le temps de calcul d'une base de Groebner est très sensible à l'ordre choisi sur les variables. Un bon choix est expérimentalement de prendre comme plus grandes variables celles des points de base, et un ordre inverse lexicographique. Mais ce n'est pas toujours le cas. Les thérèmes dans ce cas sont marqués d'une étoile dans la Figure 3.3.
- Les certificats avec des SLP sont en général meilleurs qu'avec des polynômes bruts. Par exemple, pour le théorème de Ceva, il y a une différence significative.

Les théorèmes de géométrie ne sont pas vrais dans tous les cas possibles, ils ont des conditions de non-dégénérescence : les points ne doivent pas être alignés, les droites ne doivent pas être parallellèles, etc D'un point de vue algébrique, cela signifie que l'on peut seulement prouver

$$CP = \sum_i Q_i P_i$$

où C est un polynôme en certaines variables, qui sont des paramètres du théorème. D'un point de vue logique, cela signifie que la conclusion du théorème est une disjonction de l'assertion initiale

et des cas dégénérés. Une façon de contourner cela est de travailler avec des coefficients qui sont des fractions rationnelles dans certaines variables $u_1, \ldots u_r$, appelées *paramètres*. Les polynômes ne sont plus dans $R[X_1, \ldots, X_n]$ mais dans $R(u_1, \ldots, u_r)[X_1, \ldots, X_n]$. Reprenons par exemple le théorème de Desargues. Il dit qu'étant donnés deux triangles (A, B, C) et $(A1, B1, C1)$ et un point

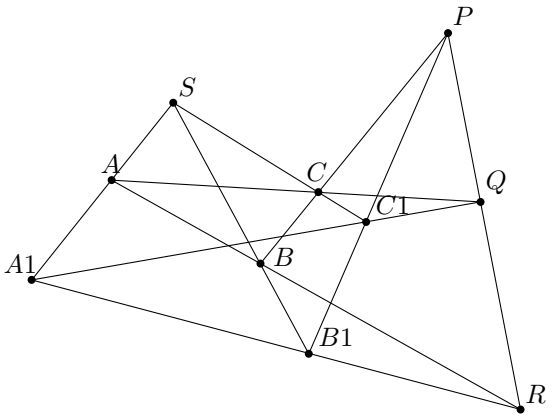

Figure 3.4: Desargues'theorem

S du plan affine tels que S, A, $A1$ sont alignés, A, B, $B1$ sont alignés, et S, C, $C1$ sont alignés, alors l'intersection R de (A, B) et $(A1, B1)$, l'intersection Q de (A, C) et $(A1, C1)$, et l'intersection P de (B, C) et $(B1, C1)$ sont aussi alignées.re also collinear. En Coq cela donne :

```
Lemma Desargues : forall A B C A1 B1 C1 P Q R S :point,
  X S = 0 -> Y S = 0 -> Y A = 0 ->
  collinear A S A1 -> collinear B S B1 -> collinear C S C1 ->
  collinear B1 C1 P -> collinear B C P ->
  collinear A1 C1 Q -> collinear A C Q ->
  collinear A1 B1 R -> collinear A B R -> collinear P Q R
  ∨ X A = X B ∨ X A = X C ∨ X B = X C ∨ X A = 0
  ∨ collinear S B C
  ∨ parallel A C A1 C1 ∨ parallel A B A1 B1.
Proof.
geo_begin.
nsatz 0%Z (X A : :X B : :Y B : :X C : :Y C : :X A1 : :Y B1 : :Y C1 : :nil)
 (X B1 : :X C1 : :Y P : :X P : :Y Q : :X Q : :Y R : :X R
   : :Y C1 : :Y B1 : :X A1 : :Y A1 : :Y C : :X C : :Y B : :X B : :nil).
Qed.
```

Le théorème est prové par deux tactiques. La première, `geo_begin`, transforme l'énoncé en égalités algébriques (les disjonctions en conclusionin deviennent des produits, les négations en hypothèses $p \neq 0$ deviennent $t * p = 1$ où t est une nouvelle variable etc). La seconde tactique, `nsatz`, prend en arguments la liste des paramètres et la liste des variables ordonnées. Elle calcule le certificat en utilisant une procédure écrite en ocaml et le vérifie ensuite.

3.5.1. *Cas dégénérés.* Sans les conditions `X A = X B ∨...∨ parallel A B A1 B1`, le théorème est faux. Pour trouver ces conditions, on tente de prouver le théorème en utilisant les variables `X A, X B, Y B, X C, Y C, X A1, Y B1, Y C1` en paramètres, i.e. en permettant de multiplier P par un polynôme en ces variables. Cela réussit et donne un coefficient c qui doit être non nul. En factorisant c, avec Maple [5] par exemple, on obtient des conditions de non-dégénerescence, mais pas forcément minimales. Dans ce cas particulier on obtient un produit de 7 facteurs simples qu'on traduit en conditions géométriques, ajoutées à la conclusion dans une disjonction.

Cela se fait manuellement pour le moment. Mais une fois trouvées, le théorème est prouvé automatiquement.

En général, ces conditions dégénérées proviennent de dénominateurs des fractions dans le certificat qui doivent être non nuls. Lorsque cela n'est pas contradictoire avec les hypothèses du théorèmes, on a bien des cas dégénérés intéressants. Dans certains exemples, cette recherche est assez coûteuse. On a du le faire pour presque tous les exemples, en prenant pour paramètres des variables bien choisies (i.e. par essai/erreur !).

Malgré tout, il y a une méthode générale, qu'il faudrait implémenter à l'avenir. L'ensemble des coefficients c tels que cP est dans l'idéal généré par $P_1, \ldots, P_s$ est lui-même un idéal. Il définit une variété algébrique qui représente les cas dégénérés où le théorème est faux. Pour complètement décrire cette variété, on doit calculer ses composantes irréductibles. Cela peut être fait encore avec un calcul de bases de Groebner. Ainsi, avec une description algébrique des cas dégénérés, on peut énoncer le théorème correctement, même si certaines conditions ne peuvent pas 6etre exprimées avec des prédicats usuels.

Toutefois, les exemples traités n'ont pas nécessité cette méthode générale, puisque l'heuristique a fonctionné.

3.5.2. *Comparaison avec d'autres systèmes.* Comparons avec deux systèmes :

Macaulay2 [9], un système dédié à la géométrie algébrique qui est très efficace pour calculer des bases de Groebner, et HOL Light [14], un assistant à la preuve qui a une tactique pour la géométrie qui utilise des bases de Groebner [15] [18] . Macaulay2, dans plusieurs cas, comme le théorème de Pascal pour le cercle, n'a pas été capable de tester l'appartenance à l'idéal (plus de 1000 secondes) car il n'a pu calculer toute la base de Groebner, alors que notre méthode réussit en 397 secondes. because it fails to compute the whole Groebner basis while our method succeeds (in 397 seconds). Pour HOL Light, la Figure 3.5 donne une comparaison plus détaillée, avec des théorèmes pour la plupart pris dans les exemples de la Figure 3.3. Comme la version de HOL Light fonctionnait en mode interprété, les temps ont été divisés par un facteur 4 pour compenser le fait que Coq utilise du code natif (4 est une bonne moyenne entre code interprété et code compilé en Ocaml).

Dans HOL Light, les théorèmes géométriques sont prouvés par réfutation, i.e. la conclusion est niée et la contradiction $1 = 0$ est prouvée en montrant que 1 est dasn l a base de Groebner. Puisque cette méthode diffère de la nôtre (le bénéfice de la division par une base partielle est perdu), chaque théorème avec des hypothèses H, un cas générique C et des cas dégénérés CP_1, …, CP_n est prouvé avec deux formulations équivalentes :

(1) les cas dégénérés sont niés en hypothèses : $H \wedge \neg CP_1 \wedge \cdots \wedge \neg CP_n \Rightarrow C$
(2) la conclusion est niée : $H \wedge \neg CP_1 \wedge \cdots \wedge \neg CP_n \wedge \neg C \Rightarrow 1 = 0$.

L'idée est que la première version profite de la division de la conclusion alors que la deuxième simule le comportement de HOL Light.

Quelques commentaires :

- le premier groupe de lignes du premier tableau contient les théorèmes pour lesquels la réfutation est plus lente que notre méthode. Pour ces théorèmes, notre tactiqueest plus rapide que HOL Light.
- le second groupe concerne les théorèmes pour lesquels la réfutation est plus rapide ; dans ce cas, notre tactique et HOL Light sont similaires.
- le second tableau contient des théorèmes écrits sous la forme $H \Rightarrow C \vee CP_1 \vee \cdots \vee CP_n$. Dans ce cas, notre méthode se comporte très mal : le polynôme représentant la conclusion est assez gros, et le diviser prend du temps.

Il n'y a pas de gagnant clair entre notre méthode et la réfutation. Toutefois il semble que pour les théorèmes de géométrie projective, mais aussi pour ceux de géométrie euclidienne , notre méthode se comporte mieux.

En moyenne, elle est plus rapide que celle de HOL Light, avec l'avantage de générer un certificat qui peut être sauvegardé et re-vérifié plus rapidement.

4. Conclusion

Peut-on faire confiance aux outils utilisés pour prouver des théorèmes ? Cette question est vieille comme les ordinateurs. La réponse est non, si on est très strict. Mais on se rapproche du oui, et les

	Temps(secondes)			
	Coq (1)	HOL Light (1)	Coq (2)	HOL Light (2)
Feuerbach	94	>2000	>2000	>2000
Ptolemé	3.7	>800	20	>800
Ceva	4.8	28	5	28
Minh	0.8	1.9	27	1.9
Butterfly	24	20	25	21
Pappus	0.9	1	1	1
Euler cercle	20	0.2	1	0.3
Pascal cercle	50	6	11	6
Simson	89	118	7.3	121
Desargues	117	28	32	29
Troispoints	3	75	2.4	75

	Temps (secondes)	
	Coq	HOL Light
Feuerbach or	1127	>2000
Ceva or	26	27
Pappus or	51	0.2
Desargues or	>500	11
Pascal circle or	44	8

Figure 3.5: Comparaison avec HOL Light

assistants de preuve en général, la théorie des types et Coq en particulier ont fait de gros progrès ! Nul doute qu'à l'avenir, la frontière entre calcul formel et preuve formelle va s'estomper, maintenant que les ordinateurs sont capables de traiter non plus seulement les calculs du mathématicien, mais aussi ses démonstrations, qui se sont le coeur de son activité. C'est sans doute l'avenir des algorithmes mathématiques d'être implémentés avec des outils de preuves formelles fondés sur les travaux de Russell, Gödel, Martin-Löf et Coquand, et non plus à l'aide de langages développés pour le commerce et la guerre...

Références

[1] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The Semantics of Reflected Proof. In *LICS*, pages 95–105. IEEE Computer Society, 1990.

[2] Samuel Boutin. Réflexions sur les quotients. *Thèse d'informatique, Université Paris VII*, 1997.

[3] Bruno Buchberger. Bruno Buchberger's PhD thesis 1965 : An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of Symbolic Computation*, 41(3-4), 2006.

[4] Amine Chaieb and Makarius Wenzel. Context Aware Calculation and Deduction. In *Calculemus/MKM*, volume 4573 of *LNCS*, pages 27–39. Springer-Verlag, 2007.

[5] Bruce W. Char, Gregory J. Fee, Keith O. Geddes, Gaston H. Gonnet, and Michael B. Monagan. A Tutorial Introduction to MAPLE. *Journal of Symbolic Computation*, 2(2) :179–200, June 1986.

[6] Shang-Ching Chou. *Mechanical geometry theorem proving*. Kluwer Academic Publishers, 1987.

[7] David Eisenbud. *Commutative Algebra : with a View Toward Algebraic Geometry*. Graduate Texts in Mathematics. Springer-Verlag, 1999.

[8] Marc Giusti, Joos Heintz, Jose Enrique Morais, Jacques Morgenstern, and Luis Miguel Pardo. Straight-line programs in geometric elimination theory. *Journal of Pure and Applied Algebra*, 124(1/3) :101–146, 1998.

[9] Daniel R. Grayson and Michael E. Stillman. Macaulay2. `http://www.math.uiuc.edu/Macaulay2/`.

[10] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[11] Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *TPHOLs 2005*, volume 3603 of *LNCS*, pages 98–113. Springer-Verlag, 2005.

[12] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. *Automated Deduction in Geometry, ADG 2008, Revised Papers, Lecture Notes in Computer Science*, 6301(2), 2011.

[13] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to Pocklington certificates in type theory. In *FLOPS'06*, volume 3945 of *LNCS*, pages 97–113. Springer-Verlag, 2006.

[14] John Harrison. HOL Light : A tutorial introduction. In *FMCAD'96*, volume 1166 of *LNCS*, pages 265–269. Springer-Verlag, 1996.

[15] John Harrison. Complex quantifier elimination in HOL. In *TPHOLs 2001 : Supplemental Proceedings*, pages 159–174. Division of Informatics, University of Edinburgh, 2001. Published as Informatics Report Series EDI-INF-RR-0046.

[16] John Harrison. Automating elementary number-theoretic proofs using Gröbner bases. In *CADE 21*, volume 4603 of *LNCS*, pages 51–66. Springer-Verlag, 2007.

[17] John Harrison. Verifying nonlinear real formulas via sums of squares. In *TPHOLs'2007*, volume 4732 of *LNCS*, pages 102–118. Springer-Verlag, 2007.

[18] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[19] Deepak Kapur. Geometry theorem proving using Hilbert's Nullstellensatz. In *SYMSAC '86 : Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*, pages 202–208. ACM, 1986.

[20] Deepak Kapur. A refutational approach to geometry theorem proving. *Artificial Intelligence*, 37(1-3) :61–93, 1988.

[21] Deepak Kapur. Automated Geometric Reasoning : Dixon Resultants, Gröbner Bases, and Characteristic Sets. In *Automated Deduction in Geometry*, volume 1360 of *LNCS*, pages 1–36. Springer-Verlag, 1996.

[22] Lawrence C. Paulson. Isabelle : A generic theorem prover. *Journal of Automated Reasoning*, 828, 1994.

[23] Loïc Pottier. Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics. In *Proceedings of the LPAR Workshops : Knowledge Exchange : Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418. CEUR Workshop Proceedings, 2008.

[24] Judit Robu. Geometry Theorem Proving in the Frame of the Theorema Project. Technical Report 02-23, RISC Report Series, University of Linz, Austria, September 2002. PhD Thesis.

[25] Laurent Théry. A Machine-Checked Implementation of Buchberger's Algorithm. *Journal of Automated Reasoning*, 26(2), 2001.

[26] Freek Wiedijk. *Formalizing 100 Theorems*. `http://www.cs.ru.nl/~freek/100`.

[27] Wen-Tsun Wu. On the Decision Problem and the Mechanization of Theorem-Proving in Elementary Geometry. In *Automated Theorem Proving - After 25 Years*, pages 213–234. American Mathematical Society, 1984.

Equipe-projet Marelle, INRIA Sophia Antipolis