Prouver les programmes : De l’ordre supérieur à Coq

Gérard Berry

Collège de France

Chaire Algorithmes, machines et langages

gerard.berry@college-de-france.fr

Cours 4, Paris, 18/03/2015

Suivi du séminaire de Christine Paulin (LRI Orsay)

COLLÈGE DE FRANCE 1530

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

Calculs des proposition et des prédicats

- Propositions élémentaires : ⊤ (vrai) , ⊥ (faux), A, B, C, ...
- Formules P construites avec des connecteurs ∧, ∨, ¬, ⇒, ⇔
- constantes, opérations, variables et fonctions dans les formules élémentaires

  $x+1 \quad f(x)+g(h(x))$

- quantificateurs ∀ et ∃ sur les variables

  $\forall x, y.\ \exists z.\ (x+z = y)$

- théories axiomatiques sur les objets

  entiers → axiomes de Peano
  ensembles → axiomes de Zermelo-Fraenkel
  etc.

- indécidable en général → assistants interactif + heuristiques

Déduction : système de Gentzen

Γ = Q, R, ... contexte = ensemble (ou liste) de formules

séquent « $\Gamma \vdash P$ » : P se démontre sous les hypothèses Γ

$$\frac{}{\Gamma \vdash \top} \qquad \frac{P \in \Gamma}{\Gamma \vdash P}$$

introduction

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

élimination

$$\frac{P \wedge Q \in \Gamma \quad \Gamma, P, Q \vdash R}{\Gamma \vdash R}$$

$$\frac{P \vee Q \in \Gamma \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R}$$

$$\frac{P \Rightarrow Q \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q \vdash R}{\Gamma \vdash R}$$

Le faux et la négation

logique intuitionniste, cf cours du 26/03/2014

$$\frac{\bot \in \Gamma}{\Gamma \vdash P}$$ *tout se prouve à partir du faux*

$$\frac{\Gamma, P \vdash \bot}{\Gamma \vdash \neg P}$$ *négation intuitionniste*
(*alternative* $\neg P =_{def} P \Rightarrow \bot$)

Attention : de $\neg(P \wedge Q)$ on ne peut pas déduire $\neg P \vee \neg Q$

$$\frac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P}$$ *preuve par l'absurde*

$\Leftrightarrow$ $$\frac{}{\Gamma \vdash P \vee \neg P}$$ *tiers exclu*

logique classique

La quantification

$$\frac{\Gamma \vdash P(x) \qquad x \text{ non libre dans } \Gamma}{\Gamma \vdash \forall x.\, P(x)} \qquad \frac{\forall x.\, P(x) \in \Gamma \qquad \Gamma, P(t/x) \vdash Q}{\Gamma \vdash Q}$$

$$\frac{\Gamma \vdash P(t/x)}{\Gamma \vdash \exists x.\, P(x)} \qquad \frac{\exists x.\, P(x) \in \Gamma \quad P(x) \vdash Q \quad x \text{ non libre dans } \Gamma, Q}{\Gamma \vdash Q}$$

- Classique : $\exists x.\, P(x) \Leftrightarrow \neg\, (\forall x.\, \neg\, P(x))$
  = raisonnement par l'absurde

- Intuitionniste : je n'accepte pas ce raisonnement !
  Pour $\exists x.\, P(x)$, je veux voir un témoin t vérifiant $P(t)$
  De $\exists x.\, P(x)$ on peut déduire $\neg(\forall x.\, \neg\, P(x))$
  mais de $\neg(\forall x.\, \neg\, P(x))$ on ne peut pas déduire $\exists x.\, P(x)$

Calcul des prédicats d'ordre supérieur

- Quantifier sur les prédicats : expressivité accrue

- Schéma d'axiomes de récurrence en calcul du premier ordre

  $(P(0) \wedge \forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$

  à instancier pour chaque P particulier → infinité d'axiomes

→ Un axiome de récurrence unique en ordre supérieur

  $\forall P.\ (P(0) \wedge \forall n.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\ P(n)$

  P est devenu une variable standard de la logique

- Mais un calcul beaucoup plus technique

  G. Huet, 1975 : algorithme pour l'unification d'ordre supérieur (indécidable en général)

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
    1. la correspondance de Curry-Howard
    2. programmer en Coq
    3. du polymorphisme aux types dépendants
    4. construire des programmes prouvés
5. Les grands succès de Coq

L'évolution des systèmes de types

- Types en LISP / Scheme : (plus chou carotte) → run-time error
- Types en C : éviter de mélanger des choux et des carottes (mais il faut parfois le faire en programmation système)
- Types en ML, Haskell etc. : un chasseur de bugs fondamental
  une aide à l'architecture

  proof = thm list → thm
  tactic = goal → (goal list × proof)

- Théorie des types de P. Martin-Löf, SystemF de J.Y. Girard (et J. Reynolds) : une nouvelle approche du calcul
- Calcul des constructions inductives, G. Huet, T. Coquand et C. Paulin, puis Coq : types dépendants et l'unification de calcul, logique et preuves

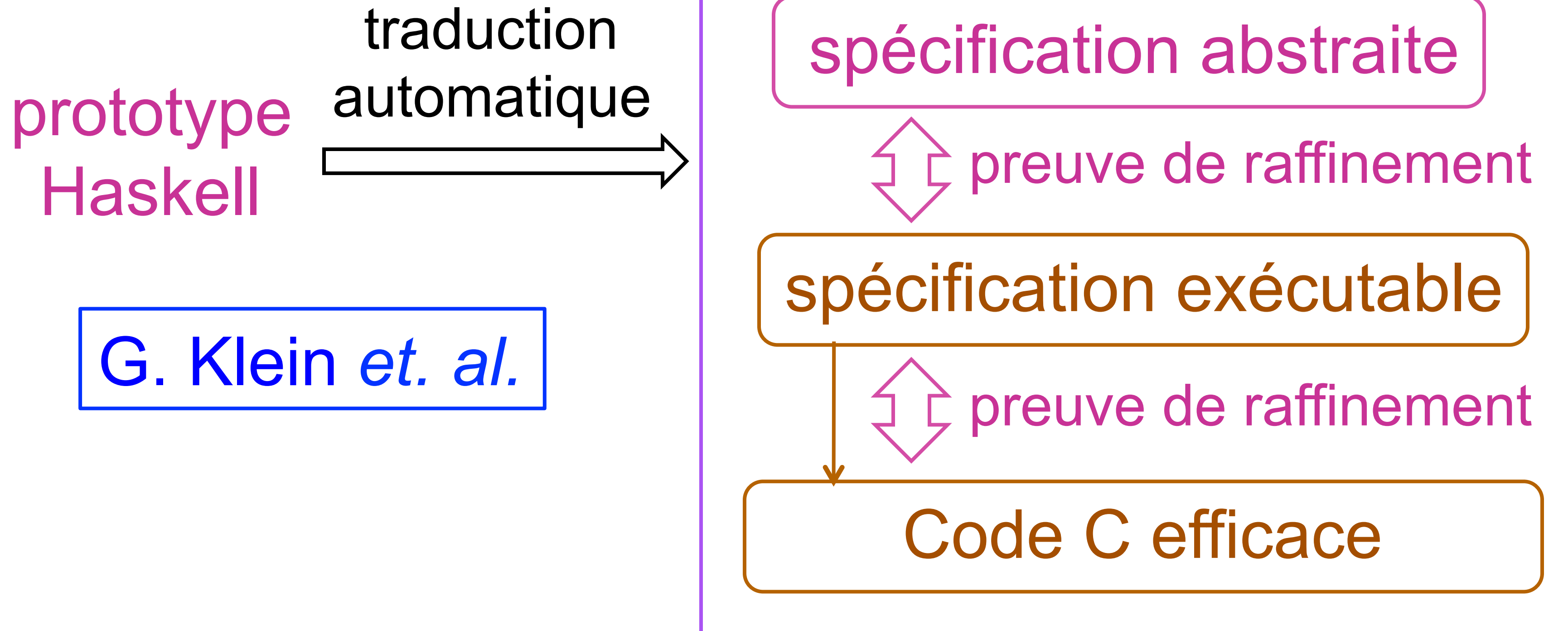
Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
    1. la correspondance de Curry-Howard
    2. programmer en Coq
    3. du polymorphisme aux types dépendants
    4. construire des programmes prouvés
5. Les grands succès de Coq

Assistants de preuve d'ordre supérieur

- HOL : M. Gordon (Cambridge)
  HOL-Light : J. Harrison (Intel Portland) :
  - λ-calcul d'ordre supérieur + théories axiomatiques + variant
  - noyau minimal : 300 lignes pour HOL-Light !
  - → arithmétique Pentium (bug de division du P.Pro, 475M$)
  - → conjecture de Kepler (oranges), Hales 2014
- PVS : S. Owre, N. Shankar, J. Rushby (SRI)
  automatisation poussée, mais pas de sûreté totale
  avionique, spatial
- Isabelle : L. Paulson (Cambridge)
  Méta-logique d'ordre supérieur
  Isabelle-HOL, Isabelle-ZF, etc.
  → maths, micronoyau seL4, etc.
- ...

Micronoyau d'OS vérifié seL4 (NICTA Sydney)



Equipe mixte OS / preuve formelle
seL4 10% plus lent que le noyau L4 standard
750,000 copies installées → industriel

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

Coq, ou l'intégration du calcul et de la preuve

- Fondations et formalisme : CoC (1984) et CiC (1989)
  - CoC = Calcul des constructions, T. Coquand et G. Huet
  - CiC = CC Inductives, C. Paulin (1989)
  - But : intégrer totalement preuve et calcul
- Racines
  - Automath (de Bruijn, 1967), automatisation des maths
  - Théorie intuitionniste des Types (Martin-Löf, 1972)
  - SystemF (Girard 1972, Reynolds 1974), λ-calcul de second ordre
  - systèmes LCF, Mentor, Centaur, etc.
- Implémentation, documentation
  - initiale : G. Huet, T. Coquand, C. Paulin,
  - actuelle : C. Murthy, H. Herbelin, JC. Filliâtre, M. Sozeau (+40)
  - livre : Y. Bertot, P. Castéran (Coq'Art)
  - guide et exemples : B. Pierce *et. al.*

Cf séminaire de G. Huet après le cours du 2 décembre 2009

Coq : les ingrédients

- Un langage global très riche (Gallina)
  - λ-calcul + système de types d'ordre supérieur hyper-puissant polymorphisme, types dépendants, etc., typage décidable
  - termes fortement normalisable : tous les calculs s'arrêtent
  - distinction programmes / preuves simplement par les types
  - grande souplesse syntaxique → notations usuelles en maths
- Pour programmer : un langage fonctionnel à la ML
  - pas de récursion générale, mais des définitions récursives gardées garantissant la terminaison
  - grande expressivité due au polymorphisme
  - modules, type classes etc. : *programming in the large*
- Pour prouver : des systèmes de tactique
  - classique
  - ssreflect (Gonthier) : interaction fine calcul / preuve

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

Une session d'assistance (en Coq)

Variables P Q R : Prop.
Theorem Imp_trans : $(P \to Q) \to (Q \to R) \to P \to R$.

Proof.

1 subgoal

$$\underbrace{(P \to Q)}_{H1} \to \underbrace{(Q \to R)}_{H2} \to \underbrace{P}_{H3} \to R$$

Une session d'assistance (en Coq)

Variables P Q R : Prop.
Theorem Imp_trans : (P→Q) → (Q→R) → P → R.

Proof.
intros H1 H2 H3.

1 subgoal

H1 : P→Q

H2 : Q→R

H3 : P

$$\frac{H1 : P \to Q \quad H2 : Q \to R \quad H3 : P}{R}$$

$\overline{H1}$ $\overline{H2}$ $\overline{H3}$

Une session d'assistance (en Coq)

Variables P Q R : Prop.
Theorem Imp_trans : $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$.

Proof.
intros H1 H2 H3.
apply H2.

1 subgoal

H1 : $P \rightarrow Q$
H2 : $Q \rightarrow R$
H3 : P

Q

Une session d'assistance (en Coq)

Variables P Q R : Prop.

Theorem Imp_trans : $(P \to Q) \to (Q \to R) \to P \to R$.

Proof.

intros H1 H2 H3.

apply H2.

apply H1.

1 subgoal

H1 : $P \to Q$

H2 : $Q \to R$

H3 : P

P

Une session d'assistance (en Coq)

Variables P Q R : Prop.
Theorem Imp_trans : $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$.

Proof.
intros H1 H2 H3.
apply H2.
apply H1.
assumption.

1 subgoal

H1 : $P \rightarrow Q$
H2 : $Q \rightarrow R$
H3 : P

P

Une session d'assistance (en Coq)

Variables P Q R : Prop.
Theorem Imp_trans : $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$.

Proof.
intros H1 H2 H3.
apply H2.
apply H1.
assumption.
Qed.

no more subgoals

H1 : $P \rightarrow Q$
H2 : $Q \rightarrow R$
H3 : P

P

La correspondance de Curry-Howard

- Une preuve (intuitionniste) de $P \rightarrow Q$ est une fonction qui prend en argument une preuve de P et renvoie une preuve de Q.
- Prouver Q par *modus ponens*, c'est appliquer une preuve de $P \rightarrow Q$ à une preuve de P pour obtenir une preuve de Q.

$$\frac{P \rightarrow Q \qquad P}{Q} \Longrightarrow \frac{f : P \rightarrow Q \qquad x : P}{f(x) : Q}$$

$\Rightarrow \; \equiv \; \rightarrow$

- Le langage naturel des preuves est le λ-calcul typé
- Le théorème prouvé est le type de la preuve

Grande unification des notions de preuve et de calcul :
En Coq, les preuves sont des objets de calcul comme les autres

La preuve de transitivité de → *n'est autre que la composition de fonctions !*

Variables P Q R : Prop.
Theorem Imp_trans : (P→Q) → (Q→R) → P → R.

Proof.
intros H1 H2 H3.
apply H2.
apply H1.
assumption.
Qed.

no more subgoals

H1 : P→Q
H2 : Q→R
H3 : P

P

Print Imp_trans.
fun (H1 : P→Q) (H2 : Q→R) (H3 : P) => H2 (H1(H3))
: (P→Q)→(Q→R)→P→R

Des types pour tous les termes

- Tout terme a un type
- Un type est un terme, donc il a un type appelé sa sorte
- Type maximal Type (en fait, hiérarchie)
- Set : Type, nat : Set, Z : Set : objets et programmes
  un programme a pour type (réalise) une spécification
- Prop : Type, les vérités logiques (théorèmes, lemmes, etc.)
  une preuve a pour type le théorème qu'elle démontre
- Program relevance : des programmes de complexité très différentes peuvent réaliser la même spécification
- Proof irrelevance : toutes les preuves pour un théorème se valent (sauf pour l'élégance)

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

Pas besoin d'axiomes, rien que des définitions !

- Les Booléens

```
Inductive bool : Set :=
   true
 | false .

Definition neg (b : bool) : bool := (* idem and, or, etc. *)
   match b with
     true => false
   | false => true
  end.
Check neg .
neg : bool → bool
```

- Principe de récurrence (trivial ici)

```
Print bool_ind .
  ∀(P:bool→Prop),
    P true → P false → (∀(b : bool), P b)
```

- Définition inductive de l'ensemble nat des entiers

```
Inductive nat : Set :=
   0
  | S : nat→nat .
```

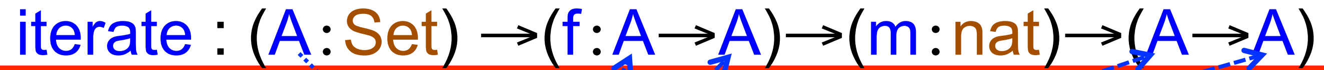
- Principe de récurrence défini automatiquement

```
Print nat_ind.
∀(P:nat→Prop),
   P 0 → (∀(n:nat), P n → P(S n)) → (∀(n:nat), P n)
```

- Définition récursive des opérations de base

```
Fixpoint plus (m n : nat) : nat :=
   match m with
    0 => n
   | S m' => S (plus m' n)
end .
Check plus .
plus : nat→nat→nat
```

Ordre supérieur et hiérarchie primitive récursive



les types dépendent d'un type argument précédent

Ordre supérieur et hiérarchie primitive récursive

iterate : (A : Set) → (f : A → A) → (m : nat) → (A → A)

iterate A f n a = $f^n(a)$

Definition plus : nat → nat → nat

:= iterate nat S

Definition mult (m n : nat) : nat

:= iterate nat (plus m) n 0

Definition exp (m n : nat) : nat

:= iterate nat (mult m) n 1

Definition superxep (m n : nat) : nat

:= iterate nat (exp m) ...

... *hiérarchie infinie primitive récursive*

Deux récurrences imbriquées → Ackermann

$$\begin{cases} Ack(0,n) = n+1 \\ Ack(m+1, 0) = Ack(m,1) \\ Ack(m+1, n+1) = Ack(m, Ack(m+1, n)) \end{cases}$$

```
Definition Ack (m : nat) : nat→nat :=
   iterate (nat→nat)
           (fun (f : nat→nat)  (p : nat) => iterate nat f (S p) 1)
           m S .
```

- Diagonalise la hiérarchie primitive récursive, m donne le nombre itérations imbriquées
- Fonctions totales définissables en Coq : celles dont la terminaison est prouvable dans l'arithmétique du second ordre (J.Y Girard, SystemF) – mais il en reste !

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

Listes polymorphes à la ML

```
Inductive list (A:Type) :=
   nil : list A
  | cons : A → list A → list A .
```

Print list_ind. (* principe de récurrence sur les listes *)

```
list_ind =
 ∀(A:Type) (P:list A → Prop) (l:list A),
  P nil →
  (∀ (a:A) (l':list A), P l' → P (cons a l') →
   P  l .
```

Listes polymorphes à la ML

```
Definition l1 := (cons nat 1 (cons nat 2 (nil nat)))
(* Inférence de type → unification *)
Arguments nil {A}. Arguments cons {A} _ _ .
Definition l2 := (cons 1 (cons 2 nil)) .
```

```
Fixpoint length {A : Type} (l : list A) → nat :=
  match  l with
    nil => 0
    cons _ l’ => S (length l’)
  end .

Eval compute in length l2.
2 : nat
```

Comptage d'occurrence d'un élément

- Demande d'avoir le test booléen d'égalité sur le domaine

```
Fixpoint count (m:nat) (l:list nat) :=
  match l with
    nil => 0
  | cons n l' => let c :=  count m l' in
                        if beq m n then S c else c end
  end .
```

Prédicats inductifs à la Prolog : le prédicat « la liste est triée pour une relation »

sorted A R l ≡ l, liste de A, est ordonnée par rapport à R

```
Inductive sorted {A : Type} (R : A→A→Prop) : list A → Prop :=
    sorted0 : sorted R nil
  | sorted1 : ∀(x:A), sorted R (cons x nil)
  | sorted2 : ∀(x y:A) (l:list A),
       R x y →
       sorted R (cons y l) →
       sorted R (cons x (cons y l))
```

Autrement dit :

sorted A R est le type des listes de A ordonnées pour R

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
    1. la correspondance de Curry-Howard
    2. programmer en Coq
    3. du polymorphisme aux types dépendants
    4. construire des programmes prouvés
5. Les grands succès de Coq

Tri d'entiers pour ≤ (le)

- Prédicat exprimant que l2 est un tri de l1 pour ≤ (le)

  Definition Sort21_le (l1 l2 : list nat) : Prop :=

  (sorted le l2) ∧ ∀(n:nat), count n l1 = count n l2 .

- Theorem uniq_sort_nat_le := (* un seul tri possible pour ≤ *)

  ∀(l1 l2 l3 : list nat),

  Sort21_le l1 l2 → Sort21_le l1 l3 → eq_list_nat l2 l3 .

- Programmation et preuve d'un algorithme de tri

  Fixpoint my_sort (l : list nat) : list nat := … .

  Theorem : my_sort_ok :=

  ∀(l : list nat), Sort21_le l (my_sort l) .

- Extraction d'un programme CAML correct

  Extraction my_sort .

Le rôle essentiel du calcul dans les preuves

- Un théorème trivial
  Theorem triv_24 : 24 = 24 . (* 24 → S (S (S (... S(0)...))) *)
  Proof . trivial . Qed .
- Une fonction bien connue

```
Fixpoint fact (n : nat) : nat :=
match n with
   0 =>  1
 | S n => mult (S n) (fact n)
end .
```

- Un théorème tout aussi trivial
  Theorem triv_fact_4 : fact 4 = 24 .
  Proof . simpl. trivial . Qed .

Ce que fait Coq, c'est de mettre toute expression en forme normale, garantie et unique (Thm. de T. Coquand)

Bibliothèques de tactiques automatiques en Coq

- intuition : preuves automatique en calcul des prédicats intuitionniste
- omega : arithmétique de Presburger sur nat et Z
- ring : égalités polynomiales sur Z et nat (sans soustraction pour nat)
- fourier : inéquation de formules linéaires sur R
- field : équation d'expressions fractionnaires sur R

Pour pouvoir ajouter une tactique automatique,
il faut qu'elle engendre pas seulement une réécriture
mais aussi sa preuve de correction (certificat)

Agenda

1. Les fondements logiques
2. Types et assistants de preuves
3. Les assistants HOL, PVS et Isabelle
4. Les principes de Coq
   1. la correspondance de Curry-Howard
   2. programmer en Coq
   3. du polymorphisme aux types dépendants
   4. construire des programmes prouvés
5. Les grands succès de Coq

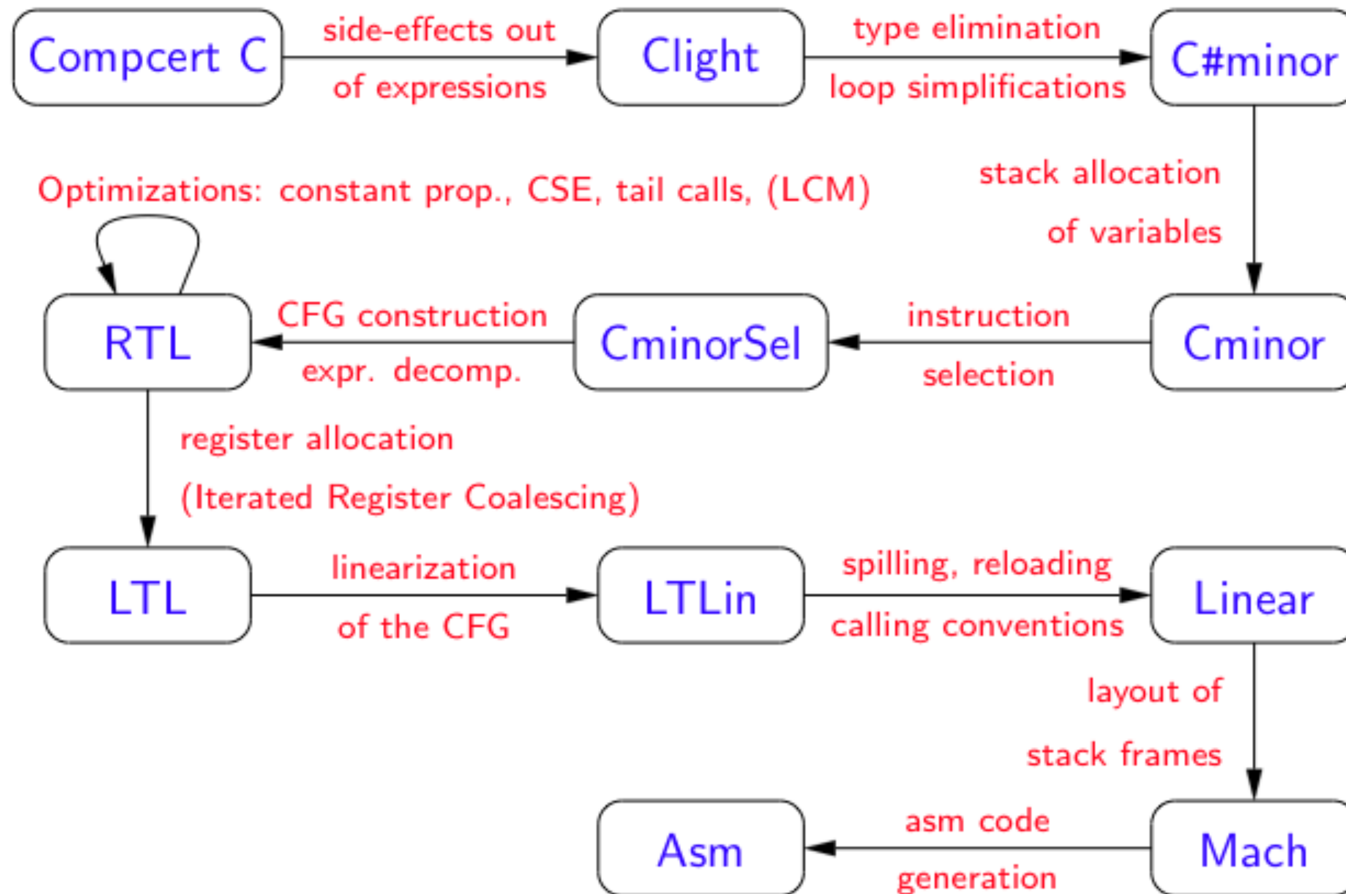
Coq, ou l'intégration du calcul et de la preuve

- Utilisateurs et contributeurs
  - progression constante, informaticiens et matheux (Princeton)
  - ex. Voevodsky. Médaille Fields (I.A.S. Priceton, homotopie)

- Des succès remarquables
  - CompCert : compilateur C prouvé, X. Leroy *et. al.*
  - théorèmes de maths : 4 couleurs et Feit-Thompson
    G. Gonthier *et. al.*

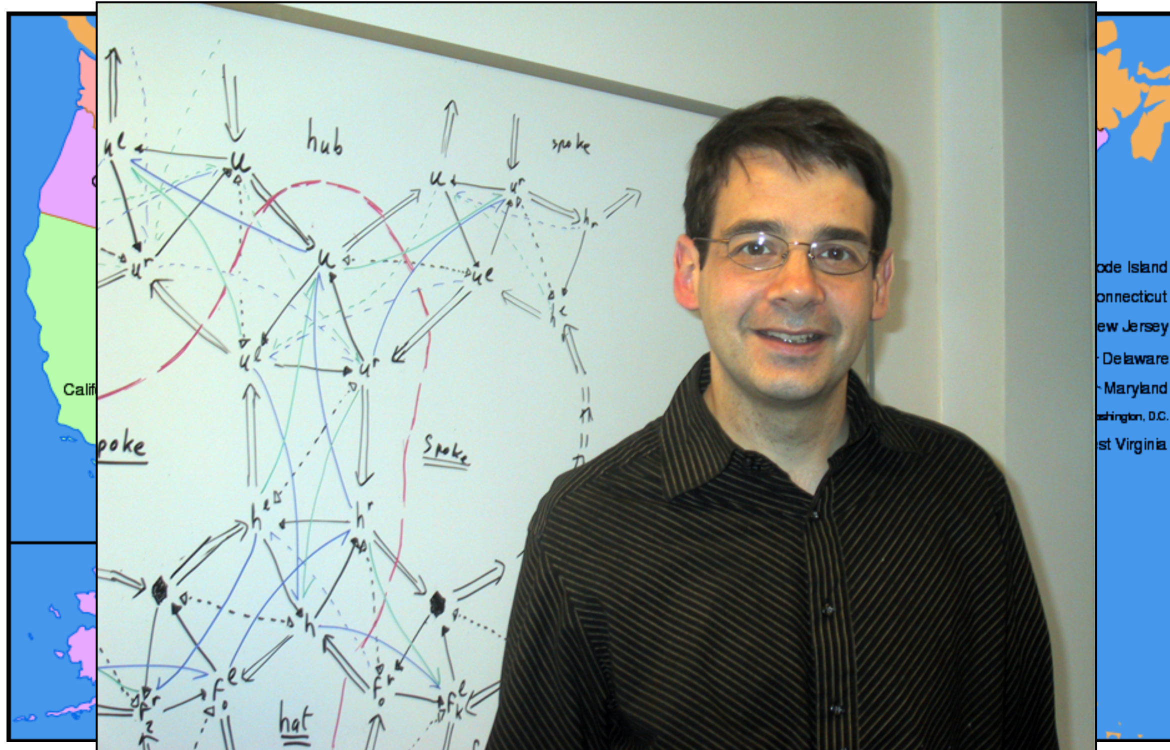
- Et des prix tout aussi remarquables
  - prix Gödel du centenaire : T. Coquand, 2008
    ACM Programming Language Software Award, 2013
    ACM Software System Award, 2014

Le compilateur vérifié CompCert (X. Leroy)



11 langages intermédiaires, 10 transformations prouvées
Utilisation d'algorithmes externes, preuve de leurs résultats

Les mathématiques sur ordinateur



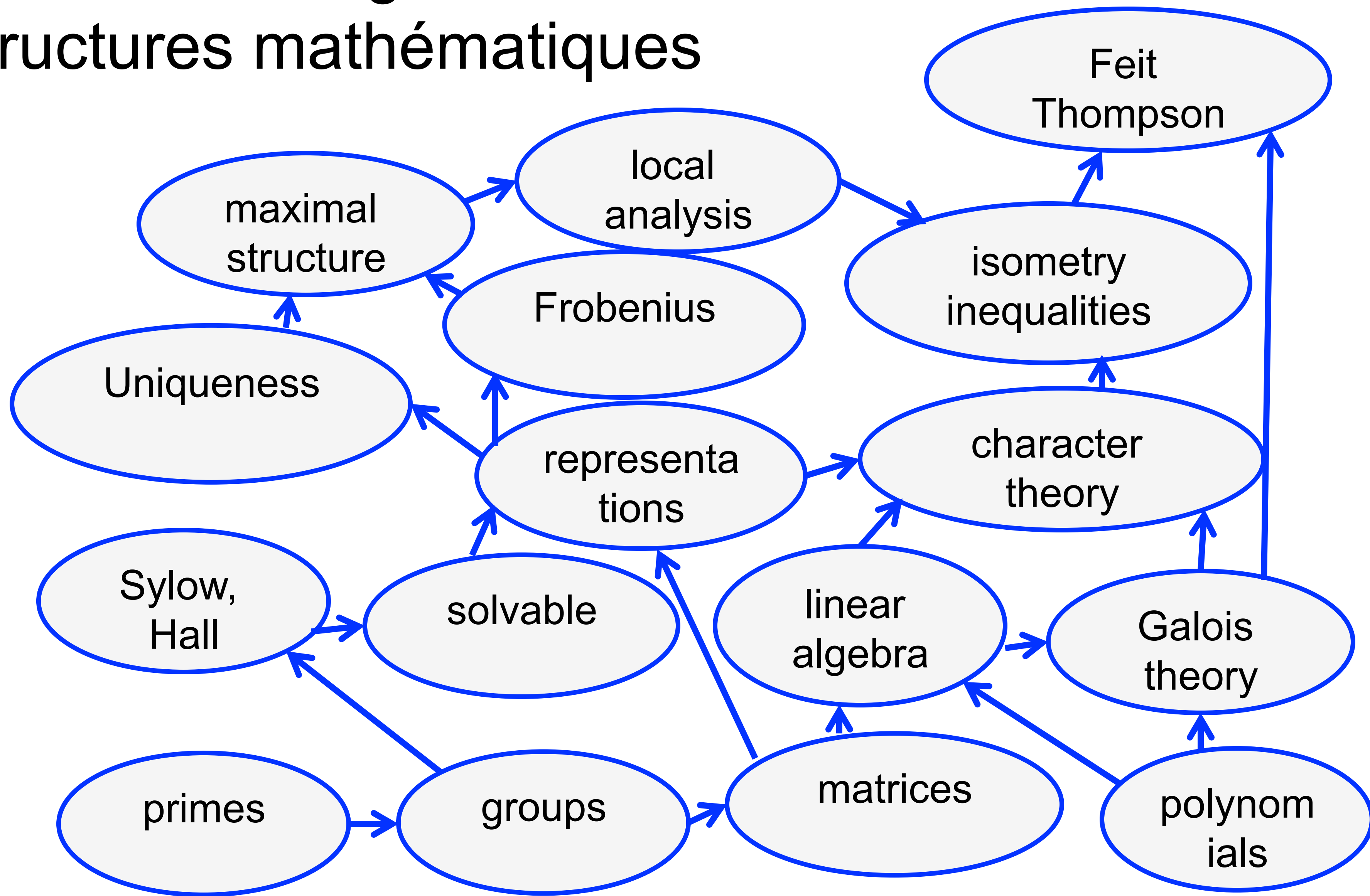
Georges Gonthier

- 1852 Guthrie
- 1976 Appel – Haken
- 2005 Gonthier (en Coq)

Preuve omise, évidente mais longue

2013 : Feit-Thompson's Odd Order Theorem
255 pages de maths lourdes → Coq !

Architecture logicielle des structures mathématiques



Du texte mathématique au texte formel

emacs@MSRC-GONTHIER

File Edit Options Buffers Tools Coq Proof-General Help

```
Proof.
pose isKi Ks M K := [&& M \in 'M_'P, \kappa(M).-Hall(M) K & Ks \subset K].
move: M K; have Pmax_sym M K X (Ks := 'C_(M`_\sigma)(K)) (Z := K <*> Ks) Mi :
    M \in 'M_'P -> \kappa(M).-Hall(M) K -> X \in 'E^1(K) -> Mi \in 'M('N(X)) ->
 [/\ Z \subset Mi, gval Mi \notin M :^: G, exists Ki, isKi Ks Mi Ki
   & {in 'E^1(Ks), forall Xs, Z \subset 'N_Mi(gval Xs)}].
- move=> PmaxM hallK E1X maxNMi.
  have [[_ maxM] [maxMi sNXMi]] := (setIdP PmaxM, setIdP maxNMi).
  have [_ [defNK defNX] [ntKs uniqCKs] _ _] := Ptype_structure PmaxM hallK.
  rewrite -/Ks in defNK ntKs uniqCKs; have [_ mulKKs cKKs _] := dprodP defNK.
  have{mulKKs} defZ: 'N_M(K) = Z by rewrite -mulKKs -cent_joinEr.
  have sZMi: Z \subset Mi.
    by rewrite -defZ; have [<- _] := defNX X E1X; rewrite setIC subIset ?sNXMi.
  have [sKMi sKsMi] := joing_subP sZMi.
  have sXMis: X \subset Mi`_\sigma by have [_ ->] := defNX X E1X.
  have sMiX: \sigma(Mi).-group X := pgroupS sXMis (pcore_pgroup _ _).
  have [q EqX] := nElemP E1X; have [sXK abelX dimX] := pnElemP EqX.
  have piXq: q \in \pi(X) by rewrite -p_rank_gt0 p_rank_abelem ?dimX.
  have notMGMi: gval Mi \notin M :^: G.
    apply: contraL (pnatPpi sMiX piXq); case/imsetP=> a _ ->; rewrite sigmaJ.
    exact: kappa_sigma' (pnatPpi (pHall_pgroup hallK) (piSg sXK piXq)).
  have kMiKs: \kappa(Mi).-group Ks.
    apply/pgroupP=> p p_pr /Cauchy[] // xs Ks_xs oxs.
    pose Xs := <[xs]>%G; have sXsKs: Xs \subset Ks by rewrite cycle_subG.
    have EpXs: Xs \in 'E_p^1(Ks) by rewrite p1ElemE // !inE sXsKs -oxs /=.
    have sMi'Xs: \sigma(Mi)^'.-group Xs.
      rewrite /pgroup /= -orderE oxs pnatE //=.
      apply: contraFN (sigma_partition maxM maxMi notMGMi p) => /= sMi_p.
      rewrite inE /= sMi_p -pnatE // -oxs andbT.
      exact: pgroupS sXsKs (pgroupS (subsetIl _ _) (pcore_pgroup _ _)).
    have uniqM: 'M('C(Xs)) = [set M] by apply: uniqCKs; apply/nElemP; exists p.
    have [x Xx ntx] := trivgPn _ (nt_pnElem EqX isT).
    have Mis_x: x \in (Mi`_\sigma)^# by rewrite !inE ntx (subsetP sXMis).
    have CMix_xs: xs \in ('C_Mi[x])^#.
      rewrite 2!inE -order_gt1 oxs prime_gt1 // inE -!cycle_subG.
      rewrite (subset_trans sXsKs) //= sub_cent1 (subsetP _ x Xx) //.
      by rewrite centsC (centSS sXsKs sXK).
    have{sMi'Xs} [|[_ _]] := pi_of_cent_sigma maxMi Mis_x CMix_xs sMi'Xs.
      by case; rewrite /p_elt oxs pnatE.
    case/mem_uniq_mmax=> _ sCxsMi; case/negP: notMGMi.
    by rewrite -(eq_uniq_mmax uniqM maxMi) ?orbit_refl //= cent_cycle.
  have{kMiKs} [Ki hallKi sKsKi] := Hall_superset (mmax_sol maxMi) sKsMi kMiKs.
  have{ntKs} PmaxMi: Mi \in 'M_'P.
    rewrite !(maxMi, inE) andbT /= -partG_eq1 -(card_Hall hallKi) -trivg card1.
    exact: subG1_contra sKsKi ntKs.
  have [_ [defNKi defNXs] _ _ _] := Ptype_structure PmaxMi hallKi.
  split=> //= [|Xs]; first by exists Ki; apply/and3P.
  rewrite -{1}[Ks](setIidPr sKsKi) nElemI -setIdE => /setIdP[E1Xs sXsKs].
  have{defNXs} [defNXs _] := defNXs _ E1Xs; rewrite join_subG /= {2}defNXs.
  by rewrite !subsetI sKMi sKsMi cents_norm ?normsG ?(centsS sXsKs) // centsC.
move=> M K PmaxM hallK /=; set Ks := 'C_(M`_\sigma)(K); set Z := K <*> Ks.
move: {2}_.+1 (ltnSn #|class_support (Z :\: (K :|: Ks)) G|) => nTG.
elim: nTG => // nTG IHn in M K PmaxM hallK Ks Z *; rewrite ltnS => leTGn.
have [maxM notFmaxM]: M \in 'M /\ M \notin 'M_'F := setDP PmaxM.
have{notFmaxM} ntK: K :!=: 1 by rewrite (trivg_kappa maxM).
have [_ [defNK defNX] [ntKs uniqCKs] _ _] := Ptype_structure PmaxM hallK.
rewrite -/Ks in defNK ntKs uniqCKs; have [_ mulKKs cKKs _] := dprodP defNK.
have{mulKKs} defZ: 'N_M(K) = Z by rewrite -mulKKs -cent_joinEr.
pose MNX := \bigcup_(X in 'E^1(K)) 'M('N(X)); pose MX := M |: MNX.
have notMG_MNX: {in MNX, forall Mi, gval Mi \notin M :^: G}.
  by move=> Mi /bigcupP[X E1X /(Pmax_sym M K)[]].
have MX0: M \in MX := setU11 M MNX.
have notMNX0: M \notin MNX by apply/negP=> /notMG_MNX; rewrite orbit_refl.
pose K_ Mi := odflt K [pick Ki | isKi Ks Mi Ki].
pose Ks_ Mi := 'C_(Mi`_\sigma)(K_ Mi).
have K0: K_ M = K.
  rewrite /K_; case: pickP => // K1 /and3P[_ /and3P[_ kK1 _] sKsK1].
  have sM_Ks: \sigma(M).-group Ks := pgroupS (subsetIl _ _) (pcore_pgroup _ _).
  rewrite -(setIid Ks) coprime_TIg ?eqxx ?(pnat_coprime sM_Ks) // in ntKs.
  exact: sub_pgroup (@kappa_sigma' M) (pgroupS sKsK1 kK1).
```

```
  have [_ [defNK defNX] [ntKs uniqCKs] _ _] := Ptype_structure PmaxM hallK.
  rewrite -/Ks in defNK ntKs uniqCKs; have [_ mulKKs cKKs _] := dprodP defNK.
  have{mulKKs} defZ: 'N_M(K) = Z by rewrite -mulKKs -cent_joinEr.
  have sZMi: Z \subset Mi.
    by rewrite -defZ; have [<- _] := defNX X E1X; rewrite setIC subIset ?sNXMi.
  have [sKMi sKsMi] := joing_subP sZMi.
  have sXMis: X \subset Mi`_\sigma by have [_ ->] := defNX X E1X.
  have sMiX: \sigma(Mi).-group X := pgroupS sXMis (pcore_pgroup _ _).
  have [q EqX] := nElemP E1X; have [sXK abelX dimX] := pnElemP EqX.
  have piXq: q \in \pi(X) by rewrite -p_rank_gt0 p_rank_abelem ?dimX.
  have notMGMi: gval Mi \notin M :^: G.
    apply: contraL (pnatPpi sMiX piXq); case/imsetP=> a _ ->; rewrite sigmaJ.
    exact: kappa_sigma' (pnatPpi (pHall_pgroup hallK) (piSg sXK piXq)).
  have kMiKs: \kappa(Mi).-group Ks.
    apply/pgroupP=> p p_pr /Cauchy[] // xs Ks_xs oxs.
    pose Xs := <[xs]>%G; have sXsKs: Xs \subset Ks by rewrite cycle_subG.
    have EpXs: Xs \in 'E_p^1(Ks) by rewrite p1ElemE // !inE sXsKs -oxs /=.
    have sMi'Xs: \sigma(Mi)^'.-group Xs.
      rewrite /pgroup /= -orderE oxs pnatE //=.
      apply: contraFN (sigma_partition maxM maxMi notMGMi p) => /= sMi_p.
      rewrite inE /= sMi_p -pnatE // -oxs andbT.
      exact: pgroupS sXsKs (pgroupS (subsetIl _ _) (pcore_pgroup _ _)).
    have uniqM: 'M('C(Xs)) = [set M] by apply: uniqCKs; apply/nElemP; exists p.
    have [x Xx ntx] := trivgPn _ (nt_pnElem EqX isT).
    have Mis_x: x \in (Mi`_\sigma)^# by rewrite !inE ntx (subsetP sXMis).
    have CMix_xs: xs \in ('C_Mi[x])^#.
      rewrite 2!inE -order_gt1 oxs prime_gt1 // inE -!cycle_subG.
      rewrite (subset_trans sXsKs) //= sub_cent1 (subsetP _ x Xx) //.
      by rewrite centsC (centSS sXsKs sXK).
    have{sMi'Xs} [|[_ _]] := pi_of_cent_sigma maxMi Mis_x CMix_xs sMi'Xs.
      by case; rewrite /p_elt oxs pnatE.
    case/mem_uniq_mmax=> _ sCxsMi; case/negP: notMGMi.
    by rewrite -(eq_uniq_mmax uniqM maxMi) ?orbit_refl //= cent_cycle.
  have{kMiKs} [Ki hallKi sKsKi] := Hall_superset (mmax_sol maxMi) sKsMi kMiKs.
  have{ntKs} PmaxMi: Mi \in 'M_'P.
    rewrite !(maxMi, inE) andbT /= -partG_eq1 -(card_Hall hallKi) -trivg card1.
    exact: subG1_contra sKsKi ntKs.
  have [_ [defNKi defNXs] _ _ _] := Ptype_structure PmaxMi hallKi.
  split=> //= [|Xs]; first by exists Ki; apply/and3P.
  rewrite -{1}[Ks](setIidPr sKsKi) nElemI -setIdE => /setIdP[E1Xs sXsKs].
  have{defNXs} [defNXs _] := defNXs _ E1Xs; rewrite join_subG /= {2}defNXs.
  by rewrite !subsetI sKMi sKsMi cents_norm ?normsG ?(centsS sXsKs) // centsC.
move=> M K PmaxM hallK /=; set Ks := 'C_(M`_\sigma)(K); set Z := K <*> Ks.
move: {2}_.+1 (ltnSn #|class_support (Z :\: (K :|: Ks)) G|) => nTG.
elim: nTG => // nTG IHn in M K PmaxM hallK Ks Z *; rewrite ltnS => leTGn.
have [maxM notFmaxM]: M \in 'M /\ M \notin 'M_'F := setDP PmaxM.
have{notFmaxM} ntK: K :!=: 1 by rewrite (trivg_kappa maxM).
have [_ [defNK defNX] [ntKs uniqCKs] _ _] := Ptype_structure PmaxM hallK.
rewrite -/Ks in defNK ntKs uniqCKs; have [_ mulKKs cKKs _] := dprodP defNK.
have{mulKKs} defZ: 'N_M(K) = Z by rewrite -mulKKs -cent_joinEr.
pose MNX := \bigcup_(X in 'E^1(K)) 'M('N(X)); pose MX := M |: MNX.
have notMG_MNX: {in MNX, forall Mi, gval Mi \notin M :^: G}.
  by move=> Mi /bigcupP[X E1X /(Pmax_sym M K)[]].
have MX0: M \in MX := setU11 M MNX.
have notMNX0: M \notin MNX by apply/negP=> /notMG_MNX; rewrite orbit_refl.
pose K_ Mi := odflt K [pick Ki | isKi Ks Mi Ki].
pose Ks_ Mi := 'C_(Mi`_\sigma)(K_ Mi).
have K0: K_ M = K.
  rewrite /K_; case: pickP => // K1 /and3P[_ /and3P[_ kK1 _] sKsK1].
  have sM_Ks: \sigma(M).-group Ks := pgroupS (subsetIl _ _) (pcore_pgroup _ _).
  rewrite -(setIid Ks) coprime_TIg ?eqxx ?(pnat_coprime sM_Ks) // in ntKs.
  exact: sub_pgroup (@kappa_sigma' M) (pgroupS sKsK1 kK1).
have Ks0: Ks_ M = Ks by rewrite /Ks_ K0.
have K_spec: {in MNX, forall Mi, isKi Ks Mi (K_ Mi)}.
  move=> Mi /bigcupP[X _ /(Pmax_sym M K)[] // _ _ [Ki Ki_ok] _].
  by rewrite /K_; case: pickP => // /(_ Ki)/idP.
have PmaxMX: {in MX, forall Mi, Mi \in 'M_'P /\ \kappa(Mi).-Hall(Mi)(K_ Mi)}.
  by move=> Mi /setU1P[-> | /K_spec/and3P[]//]; rewrite K0.
have ntKsX: {in MX, forall Mi, Ks_ Mi != 1}.
  by move=> Mi /PmaxMX[MX_Mi /Ptype_structure[] // _ _ []].
```

--(Unix)-- BGsection14.v   51% L1374 SVN-4643 (coq Holes)--------

--(Unix)-- BGsection14.v   51% L1374 SVN-4643 (coq Holes)--------

Switch to buffer in other window (default *scratch*):

Conclusion

- Développés à bas bruit depuis les années 1970, les assistants de preuve deviennent des outils importants, en informatique et maths
- Leurs succès pratiques dépassent maintenant les prévisions initiales les plus optimistes
- Leur utilisation reste encore complexe, mais des formations sont en cours à bonne échelle (merci à Y. Bertot, P. Castéran, B. Pierce etc.)

Coq'Art : http://www.labri.fr/perso/casteran/CoqArt/

Software Foundations : http://www.cis.upenn.edu/~bcpierce/sf/current/index.html

Travail en cours par Lionel Rieg au Collège de France :
preuve et extraction d'un compilateur d'Esterel en circuits
Bientôt expliqué en cours ?