L'assistant de preuves Coq

Pierre Castéran, Université de Bordeaux et LaBRI

JDEV 2015, 3 juillet 2015

coq.inria.fr

De 1985 à 2015, de la théorie aux applications

- 1985 : CoC, vérificateur de preuves écrit en CaML.
- 2003 : Vérification avec Coq des propriétés d'isolation de l'applet JavaCard.
- 2004 : Preuve formelle du théorème des quatre couleurs.
- 2008 Première version publique (1.2) du compilateur C Compcert.
- 2013 Bedrock : Bibliothèque de raisonnement sur des programmes de bas-niveau.
- 2013, 2014 : Prix ACM Sigplan, puis ACM "Software System Award 2013"
- 2014 : Preuve du théorème de Feit-Thomson
- Actuellement : Bibliothèques Certicrypt, HOTT, etc.

Programming Languages Software Award, *et* Software System Award

Attribué par l'ACM à une institution ou des personnes en reconnaissance du développement d'un logiciel qui a eu un impact significatif sur la recherche, l'implantation et l'environnement des langages de programmation. Cet impact peut se refléter dans l'adoption par une grande communauté au sein de projets de recherche, dans la communauté open-source, ou dans des applications commerciales.

awards.acm.org/software_system/

2013 : The Coq proof assistant

Coq sur Youtube :
www.youtube.com/watch?v=vwA4JZ-5qMU

The Coq proof assistant provides a rich environment for interactive development of machine-checked formal reasoning. Coq is having a profound impact on research on programming languages and systems, making it possible to extend foundational approaches to unprecedented levels of scale and confidence, and transfer them to realistic programming languages and tools. It has been widely adopted as a research tool by the programming language research community, as evidenced by the many papers at conferences whose results have been developed and/or verified in Coq.

It has also rapidly become one of the leading tools of choice for teaching the foundations of programming languages, with courses offered by many leading universities and a growing number of books emerging to support them. Last but not least, these successes have helped to spark a wave of widespread interest in dependent type theory, the richly expressive core logic on which Coq is based.

As a software system, Coq has been in continuous development for over 20 years, a truly impressive feat of sustained, research-driven engineering. The Coq team continues to develop the system, bringing significant improvements in expressiveness and usability with each new release.

In short, Coq is playing an essential role in our transition to a new era of formal assurance in mathematics, semantics, and program verification.

Développement logiciel : les maillons faibles

1. Définition d'un problème à résoudre : ambiguïté du semi-formel
2. Choix d'un algorithme : correction de l'algorithme, conditions d'application
3. Transcription dans un langage de programmation : adaptation aux contraintes du langage, choix de la représentation des données, erreurs de transcription manuelle, etc.
4. Compilation : bugs dans le compilateur (souvent dans la phase d'optimisation du code)
5. Exécution : bugs dans le matériel (processeur)
6. Bouclage, erreurs à l'exécution : déréférencement de pointeurs, erreurs arithmétiques, etc.

Preuve de programmes par l'exemple

Nous présentons quelques exemples simples sur le rôle que peuvent jouer des assistants de preuve dans le développment de programmes corrects.

- Programmation fonctionnelle :
  Ce paradigme, permet, non seulement de construire des composants facilement certifiables et d'efficacité raisonnable , mais aussi d'exprimer des spécifications formelles.
- Programmation impérative :
  Nous aborderons les relations pouvant exister entre, par exemple, la programmation en C et l'univers de la logique et des langages fonctionnels.

Programmation en *Coq*

Prenons un problème très simple ...

Calculer x^n.

Remarques

- Ce problème a l'avantage d'être simple à énoncer, sans compter sur des applications en cryptologie, où l'exposant n peut être relativement grand.
- Cette simplicité nous permettra de nous concentrer sur les méthodes de certification d'algorithmes.

Voir `demo1.v`

Quels sont les défauts de la fonction présentée ?

```
Function expt (x:Z)(n:nat) :=
match n with
| O => 1
| S p => x * expt x p
end.
```

- La définition est monomorphe : on calcule les puissances d'un nombre entier en précision illimitée, mais pas d'entiers machine, de matrices, etc.
- Elle est très inefficace, pour des exposants pouvant être grands (complexité linéaire)

Polymorphisme

L'exponentiation est définie pour un type quelconque *A* muni d'une "multiplication" *op* et d'une "unité" *one*. Voir demo2.v

```
Variables (A:Type)
          (op : A -> A -> A)
          (one : A).
Notation ``x * y'' := (op x y).

Function expt (x:A) (n: nat) :=
match n with
| O => one
| S p => x * (expt x p)
end.
```

Une version plus efficace

L'algorithme d'exponentiation rapide est basé sur la représentation binaire de l'exposant et les égalités suivantes :

$$
\begin{align}
x^1 &= x \tag{1}\\
x^{2p} &= (x^2)^p \tag{2}\\
x^{2p+1} &= (x^2)^p \times x \tag{3}\\
x^1 \times a &= x \times a \tag{4}\\
x^{2p} \times a &= (x^2)^p \times a \tag{5}\\
x^{2p+1} \times a &= (x^2)^p \times (a \times x) \tag{6}
\end{align}
$$

```
Function bin_expt_aux (x acc:A)(p:positive) : A
  (* acc * (x ^ p) *) :=
match p with
    | 1 => acc * x
    | q~0 => bin_expt_aux (x * x) acc q
    | q~1 =>  bin_expt_aux (x * x) (acc * x) q
end.

Function bin_expt  (x:A)(p:positive) :=
match p with
  | 1 => x
  | q~0 => bin_expt  (x * x) q
  | q~1 => bin_expt_aux  (x * x) x q
end.
```

Bilan provisoire

On dispose de deux fonctions pour calculer des puissances entières :

- Les deux fonctions sont polymorphes ☺
- La fonction expt est vraiment inefficace ☹
- La fonction bin_expt est logarithmique en son exposant ☺
- Le code d'expt est simple et compréhensible par tous ☺
- Le code de bin_expt est plus complexe et peut être erroné ☹.

Bilan provisoire

On dispose de deux fonctions pour calculer des puissances entières :

- Les deux fonctions sont polymorphes ☺
- La fonction expt est vraiment inefficace ☹
- La fonction bin_expt est logarithmique en son exposant ☺
- Le code d'expt est simple et compréhensible par tous ☺
- Le code de bin_expt est plus complexe et peut être erroné ☹.
- Il faut garder les deux fonctions dans le même environnement de travail, comme deux vues d'une même fonctionnalité. Ces deux vues seront reliées par des certificats.

- La fonction naïve expt sert de spécification, de référence,
- La fonction bin_expt s'utilise lors de calculs, ou peut être traduite vers d'autes langages de programmation.
- On écrit à l'aide de *Coq* une preuve de correction de l'implantation bin_expt par rapport à expt.

$$\forall (n : \texttt{positive})(x : A), \mathit{bin_exp}\ x\ n = \ x^n$$

Un exemple de preuve simple

Théorème :

$$\forall x\, n\, p,\ x^{n+p} = x^n \times x^p$$

Attention aux pré-suppositions implicites !

Cette propriété, ainsi que la correction de bin_expt, requiert deux hypothèses :

- La multiplication est associative
- La constante one est élément neutre pour la multiplication.

Voir `demo5.v`

Avec un peu d'efforts, on arrive à prouver un théorème très général[1].

```
Theorem bin_exp_correct :
∀ (A:Type)(op: A -> A -> A)(one:A),
  Monoid op one ->
  ∀ (p:positive) (x:A), bin_expt x p = x ^ p.
```

On obtient donc deux vues de x^n : une pour calculer, l'autre pour raisonner. Le théorème bin_exp_correct sert de passerelle entre ces deux vues.

1. Avec quelques abus de notation pour simplifier l'exposé

On peut alors dériver des propriétés de l'implantation à partir de propriétes de la spécification.

```
bin_expt x (n + p) = (* correction de bin_exp *)
x ^ (n + p) = (* propriété de expt *)
x ^ n * x ^ p = (* correction de bin_exp *)
bin_expt x n * bin_expt x p
```

En résumé...

- Nous disposons d'une définition de référence lisible, pas forcément efficace pour une fonction donnée,
- On écrit un algorithme efficace pour la fonction,
- On prouve formellement que les deux fonctions renvoient toujours le même résultat,
- Cette preuve peut faire apparaître des hypothèses, parfois implicites dans les descriptions sur papier
- la version certifiée peut être traduite vers un autre langage.

Allons plus loin ...

- Considérons un peu notre algorithme d'exponentiation binaire. Par exemple, pour calculer x^{87}, la suite de puissances de x suivante est calculée :
  $x, x^2, x^3, x^4, x^7, x^8, x^{16}, x^{23}, x^{32}, x^{64}, x^{87}$
  Soit 10 multiplications.

Allons plus loin ...

- Considérons un peu notre algorithme d'exponentiation binaire. Par exemple, pour calculer x^{87}, la suite de puissances de x suivante est calculée :
  $x, x^2, x^3, x^4, x^7, x^8, x^{16}, x^{23}, x^{32}, x^{64}, x^{87}$
  Soit 10 multiplications.
- Or, on peut mieux faire ! $x, x^2, x^3, x^6, x^7, x^{10}, x^{20}, x^{40}, x^{80}, x^{87}$
  Soit 9 multiplications.
- Comment construire et valider des algorithmes plus efficaces que l'exponentiation binaire ?

La suite de notre exemple illustre sur une étude de cas très simple des techniques réellement utilisées en *Coq* dans le domaine des langages de programmation : syntaxe, sémantique, compilation, preuves de correction.

On travaille sous deux hypothèses :

- Le coût d'un calcul de x^n est proportionnel au nombre de mutiplications effectuées,
- Ce calcul se réduit à une suite de multiplications

On définit un micro-langage de programmation spécialisé dans ce type de calcul. Les programmes prennent la forme de suites de multiplications, appelées chaîne d'additions depuis Brauer[1939].

```
(** calcul de x ^ 87 *)

Example C87 (A:Type)(x:A) :=
  x2 <--- x * x ;  x3 <--- x2 * x ;
  x4 <--- x3 * x3 ;  x5 <--- x4 * x ;
  x6 <--- x5 * x3 ;  x7 <--- x6 * x6 ;
  x8 <--- x7 * x7 ;  x9 <--- x8 * x8 ;
  x10 <--- x9 * x5 ;
  Return x10.
```

Problèmes posés

1. Vérification qu'un tel programme calcule bien la puissance annoncée
2. Construction d'un générateur automatique de tels programmes
3. Validation des programmes engendrés par ce générateur

Réutilisation de démonstrations

- L'écriture, même assistée, de démonstrations, est une activité chronophage et pas toujours couronnée de succès.
- On essaie de privilégier deux méthodes :
  - Utiliser les calculs (automatiques) dès que possible.
  - Prouver les thèorèmes les plus abstraits/génériques possible.
- Notons que nous retrouvons des considérations similaires en programmation (*cf.* les classes abstraites des langages à objets.)

Génération certifiée de programmes

- Au lieu de prouver la correction d'un programme d'exponentiation pour un exposant n donné, on peut envisager de construire un générateur de tels programmes, et de prouver une fois pour toutes que ce générateur construit un algorithme d'exponentiation pour tout exposant :

```
Definition correct_algo (c: chain)(p:positive) :=
  forall A ... x, execute (c x) = x ^ p.

Definition correct (gen : positive -> chain) :=
  forall n, correct_algo (gen c p) p.
```

Par exemple, nous avons certifié la construction automatique d'algorithmes meilleurs que l'exponentiation binaire.

```
Theorem OK : correct my_chain_generator.
 (* Proof skipped *)

Lemma L : forall x,
  execute (my_chain_generator 87) = x ^ 87.
Proof.
  intro x; apply OK.
Qed.
```

Que nous apprend l'exemple précédent ?

- *Coq* permet d'écrire des programmes, de les exécuter aux fins de tests, d'en prouver des propriétes et leur correction.
- la preuve d'un thèorème (propriété mathématique ou correction de programme) se fait de façon interactive :
  - Par l'intermédiaire de tactiques (*e.g.* intro, induction, simpl, etc.) , le système nous aide à construire une preuve complète,
  - L'utilisateur et le concepteur de bibliothèques pour *Coq* disposent d'un langage permettant de construire des tactiques élaborées pour des domaines spécifiques.
- Avant l'enregistrement d'un théorème, sa preuve est vérifiée à l'aide d'un algorithme connu.

- Sur un exemple jouet, nous avons vu comment :
  - définir la syntaxe d'un langage de programmation
  - en définir la sémantique
  - prouver des propriétés (y compris la correction) de programmes écrits dans ce langage
  - étudier et valider des générateurs de programmes corrects par construction.

Points non abordés

- Possibilité d'écrire simultanément le programme et sa preuve, en partant d'une spécification.
- Systèmes de modules, classes de types,
- Plusieurs bibliothèques de programmes certifiés : bibliothèque standard, ssreflect, Color, etc.
- Certains aspects sont plus difficiles que d'autres : objets infinis, types dépendants, etc.

Coq et la programmation impérative

Les exemples montrés jusqu'à présent se sont limités à la programmation purement fonctionnelle.
Qu'en est-il de la programmation impérative (par exemple en `C`) ?

Coq reste un langage de programmation fonctionnelle permettant la preuve de théorèmes. Son utilisation pour la programmation “du monde réel” se situe entre autres dans les axes suivants :

- Certification d'outils pour la programmation :
- Spécification formelle de programmes

Exemples d'applications

- Compilateur `C` certifié CompCert
- Analyse statique de programmes `C` Verasco
- Vérification de programmes truffés d'annotations : Frama-C/Jessie
  - validation de la génération de conditions de vérification
  - preuve des conditions de vérification complexes
- Vérification de constructions cryptographiques : encryption, signatures numériques, etc.
  www.easycrypt.info/trac/

CompCert : un compilateur pour C validé par *Coq*

The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software. Such verified compilers come with a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program. By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs.

The CompCert C verified compiler is a high-assurance compiler for almost all of the ISO C90 / ANSI C language, generating efficient code for the PowerPC, ARM and x86 processors.

compcert.inria.fr

Verasco : Un analyseur statique pour C formellement vérifié

- Actuellement à l'état de prototype dévéloppé par des membres du projet `CompCert`.
- Cet analyseur permet de garantir l'absence de comportement indéfini dans un code écrit en `C99` sans allocation dynamique ni récursion.
- Il repose sur des techniques d'analyse statique par interprétation abstraite afin de calculer les invariants de boucles, notamment.
- Il prend en compte toutes les structures de contrôle disponibles : boucles, tests, gotos, appels de fonctions, etc.

compcert.inria.fr/verasco/

Preuve d'un programme C avec Frama-C/Jessie

```
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
  while (low <= high) {
    int middle = (low + high) / 2;
    if (t[middle] < v)
       low = middle + 1;
    else if (t[middle] > v)
     high = middle -1;
    else return middle;
 }
 return -1;
}
```

frama-c.com/jessie.html

Premier diagnostic (Propriétés de sûreté)

```
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
  while (low <= high) {
    int middle = (low + high) / 2;
    if (t[middle] < v)
       low = middle + 1;
    else if (t[middle] > v)
     high = middle -1;
    else return middle;
 }
 return -1;
}
```

Premier ajoût : Pré-conditions et invariants de boucle

```
//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
//@loop invariant 0 <= low && high <= n-1;
  while (low <= high) {
    int middle = (low + high) / 2;
    if (t[middle] < v)
       low = middle + 1;
    else if (t[middle] > v)
     high = middle -1;
    else return middle;
 }
 return -1;
}
```

```
//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
//@loop invariant 0 <= low && high <= n-1;
  while (low <= high) {
    int middle = (low + high) / 2;
 ...
```

Arithmetic Overflow ☹.

gWhy: a verification conditions viewer

File Configuration Proof

Proof obligations	Alt-Ergo 0.94	Statistics
2. loop invariant initially holds		
3. loop invariant preserved		
4. loop invariant preserved		
5. loop invariant preserved		
6. loop invariant preserved		
7. postcondition		
8. postcondition		
Function binary_search Behavior `success'		2/2
1. postcondition		
2. postcondition		
Function binary_search Safety		16/17
1. check arithmetic overflow		
2. check arithmetic overflow		
3. check arithmetic overflow		
4. check arithmetic overflow	?	
5. check division by zero		
6. check arithmetic overflow		
7. check arithmetic overflow		
8. pointer dereferencing		
9. pointer dereferencing		

```
  offset_min(intP_a_2_alloc_table, a_0) <= 0 and
  offset_max(intP_a_2_alloc_table, a_0) >=
integer_of_int32(size) - 1)
result: int32
H7: integer_of_int32(result) = 0
low: int32
H8: low = result
H9: -2147483648 <= integer_of_int32(size) - 1 and
    integer_of_int32(size) - 1 <= 2147483647
result0: int32
H10: integer_of_int32(result0) = integer_of_int32
(size) - 1
high: int32
H11: high = result0
high0: int32
low0: int32
H12: true
H13: (0 <= integer_of_int32(low0) and
     integer_of_int32(high0) <= integer_of_int32
(size) - 1)
H14: integer_of_int32(low0) <= integer_of_int32
(high0)
________
integer_of_int32(high0) + integer_of_int32(low0) <=
```

```
  @  \forall integer k; 0 <= k < size && a[k] ==
target ==> low <= k <= high;
   @ loop variant high-low;
   @*/
  while (low <= high ) {
    int middle = (high + low) / 2;
    //@ assert low <= middle <= high;
    if (target < a[middle])
      high = middle - 1;
    else if (target > a[middle])
      low = middle + 1;
    else
      return middle;
```

Timeout 10 | Pretty Printer | file: binary_acsl.c VC: check arithmetic overflow

Modification du code

```
//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
//@loop invariant 0 <= low && high <= n-1;
  while (low <= high) {
    int middle = low + (high - low) /2;
 ...
```

Absence de bouclage

```
//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
//@ loop invariant 0 <= low && high <= n-1;
//@ loop variant high - low;
  while (low <= high) {
    int middle = low + (high - low) /2
    if (t[middle] < v)
       l = middle + 1;
    else if (t[middle] > v)
     high = middle -1;
    else return middle;
 }
 return -1;
}
```

Spécification fonctionnelle

```
/*@ requires n >= 0 && \valid_range(t,0,n-1)
 @
 @ behavior success:

 @ assumes // array t is sorted in increasing order
 @ \forall integer k1, k2; 0 <= k1 <= k2 <= n-1
 @    ==> t[k1] <= t[k2];
 @ assumes // v appears somewhere in the array t
 @ \exists integer k; 0 <= k <= n-1  && t[k] == v;
 @ ensures 0 <= \result n-1 && t[\result] == v
 @
```

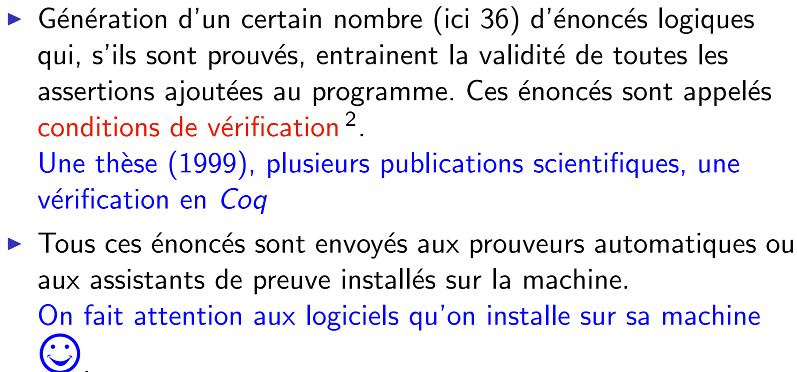
Spécification fonctionnelle (suite)

```
  @ behavior failure:
  @ assumes  // v does not appear anywhere in the array t
  @ \forall integer k; 0 <= k <= n - 1 ==> t[k] != v;
  @ ensures \result == -1;
 */
  int binary_search(long t[], int n, long v) {
  ...
```

Version finale (prouvée automatiquement)

```
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
/*
   @ loop invariant 0 <= low && high <= n-1;
   @ for success:
   @ loop invariant
   @ \forall integer k;
      0 <= k < n && t[k] == v ==> low <= k <= high
   @ loop variant high - low;
/*
  while (low <= high) { ...
 }
 return -1;
}
```

Peut-on se fier à ce genre de preuve automatique ?



`Frama-C` procède en deux temps :

- Génération d'un certain nombre (ici 36) d'énoncés logiques qui, s'ils sont prouvés, entrainent la validité de toutes les assertions ajoutées au programme. Ces énoncés sont appelés conditions de vérification [2].
  Une thèse (1999), plusieurs publications scientifiques, une vérification en *Coq*
- Tous ces énoncés sont envoyés aux prouveurs automatiques ou aux assistants de preuve installés sur la machine.
  On fait attention aux logiciels qu'on installe sur sa machine ☺.

2. *c.f.* les obligations de preuves de la méthode B

Utilité de *Coq* dans ce type de développement

- *Coq* permet de valider la génération de conditions de vérification, et donc de garantir que, si elles sont toutes valides, alors toutes les annotations seront vraies au cours de l'exécution.
- D'un point de vue plus concret, un assistant à la preuve interactive permet de vérifier les conditions de vérification laissées non résolues par les prouveurs automatiques.

Les assistants à la preuve sont ils fiables ?

- D'après Thomas Hales, il y aurait dans tout programme un bug toutes les 500 lignes de code.
- Par exemple, la distribution des sources de *Coq* mesure 4 Mo (version comprimée) : code `C`, `Ocaml`, `Coq`
- On ne peut pas espérer une absence totale de bugs !
- Sans compter les compilateurs : `Ocaml`, `C` ...

Ce qui fonde la confiance dans une preuve formelle, c'est que l'ensemble de l'assistant de preuve est contrôlé par une toute petite partie, qu'on appellera le noyau du logiciel.
La taille du noyau est suffisamment petite (500 lignes environ) pour pouvoir être scrutée par un humain ; de plus, il a été (at)testé sur un grand nombre de situations variées.
Même s'il n'est pas capable de se certifier lui-même (théorème d'incomplétude de Gödel oblige), le noyau est capable de certifier le reste de l'assistant de preuve.
images.math.cnrs.fr/Coq-et-caracteres.html

Et l'automaticité ?

- Les démonstrateurs automatiques de théorèmes sont d'une minipulation plus aisée que les assistants de preuve interactifs
- Mais ils sont limités (résultats d'indécidabilité)

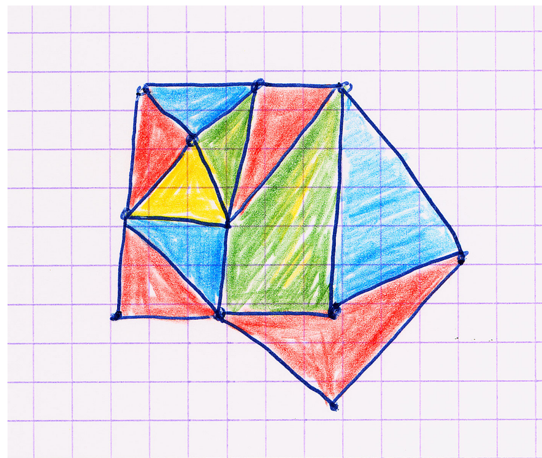
Et l'automaticité ?

- Les démonstrateurs automatiques de théorèmes sont d'une minipulation plus aisée que les assistants de preuve interactifs
- Mais ils sont limités (résultats d'indécidabilité)
- Les assistants à la preuve contiennent des procédures de semi-décision : preuve automatique ou time-out, pour certains types de problèmes.
- En amont, des outils comme Frama-C, Why3, Rodin, envoient leurs obligations de preuve en priorité aux prouveurs automatiques. Seules les obligations non résolues sont traitées de façon interactive.

Coq et les mathématiques (s'il reste du temps ☺)

- Théorème des quatre couleurs
- Feit-Thomson
- Modélisation de l'analyse
- Logique, ordinaux
- HOTT
- ...

Des preuves illisibles ?



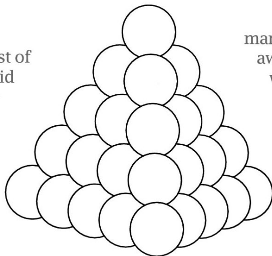
Le théorème des quatre couleurs

- Conjecture proposée en 1852
- Beaucoup de « preuves » fausses, y compris par de grands mathématiciens (de Morgan, Cayley, Hamilton),
- 1976, Appel et Haken :Première preuve sur ordinateur ; des milions de cas traités par programme, justifiés par une analyse manuelle gigantesque (1936 cas, 400 pages),
- 1995, Robertson, Saunders, Seymour et Thomas : simplification en 1995,
- 2005 : Preuve en *Coq* par Gonthier : 60000 lignes de commandes, mais partie critique réduite : 200 lignes de définitions, vérificateur de preuves de *Coq*.

La concurrence : Conjecture de Kepler

- Conjecture de Kepler (1611) : *Il n'existe pas de façon de ranger des sphères de même diamètre dont la densité soit supérieure à celle du rangement cubique à faces centrées (empilements d'oranges ou de boulets de canon) :* $\frac{\pi}{\sqrt{18}}$
- Contributions de Gauss, Thue (en 2D), Fejes Tóth.

When Hilbert introduced his famous list of 23 problems, he said a test of the perfection of a mathematical problem is whether it can be explained to the first person in the street. Even after a full century, Hilbert's problems have never been thoroughly tested. Who has ever chatted with a telemarketer about the Riemann hypothesis or discussed general reciprocity laws with the family physician?



market. "We need you down here right away. We can stack the oranges, but we're having trouble with the artichokes."

To me as a discrete geometer there is a serious question behind the flippancy. Why is the gulf so large between intuition and proof? Geometry taunts and defies us. For example, what about stacking tin cans? Can anyone doubt that parallel rows of upright cans give the best arrangement? Could some disordered heap of cans

- Preuve par Thomas Hales (1998), dont certaines parties utilisent des calculs sur machine :
  - Énumération de graphes (planaires) pouvant être des contre-exemples à la conjecture. Ces contre-exemples éventuels sont caractérisés par 8 contraintes portant sur les cycles, degrés des sommets, affectation de poids aux faces. Le programme *Java* de Hales énumère 5128 graphes satisfaisant ces contraintes.
  - Programmation linéaire, destinée à vérifier qu'aucun de ces graphes ne constitue un réel contre-exemple.

Le projet Flyspeck

- Vérification de la preuve de Hales par une équipe de 12 arbitres, et conférence consacrée à la preuve : résultat : certitude à 99%,
- Projet Flyspeck : construire une version formelle de la preuve de 1998, principalement à l'aide principalement de *HOL/Light* : mais aussi en *Isabelle/HOL* et *Coq*.

Contribution de Bauer et Nipkow (2005)

- Preuve de la complétude de l'énumération des graphes par le programme de Hales,
- Détection de redondances (2771 graphes au lieu des 5128 énumérés par le programme *Java* de Hales,)
- Détection d'une contrainte absente de la preuve de 98, mais traitée dans le programme *Java*,
- Bilan : la mécanisation de cette preuve a permis de nombreuses simplifications dans le calcul de l'énumération des graphes,
- 17000 lignes d'*Isabelle*, 165 minutes de vérification, 2300000 graphes engendrés durant la preuve
- Reste à faire : vérifier que les 2771 graphes ne sont pas des contre-exemples réels à la conjecture de Kepler.

Le *Théorème* de Kepler

Été 2014 : La preuve est complète.

En guise de conclusion ...

Pour vérifier des programmes complexes[3] ou prouver des théorèmes incertains, on peut utiliser une nouvelle sorte d'outils, les *assistants à la preuve*.

- Aide à la construction de démonstrations complexes (programmable)
- vérification de *toutes* les étapes.

Effort considérable, mais tout dépend des enjeux.

3. Applications industrielles : sécurité, vérification de circuits, etc.

Le mot de la fin

- But what actually happened was this : Voevodsky told mathematicians that their lives are about to change. Soon enough, they're going to find themselves doing mathematics at the computer, with the aid of computer proof assistants.
- Soon, they won't consider a theorem proven until a computer has verified it.
- Soon, they'll be able to collaborate freely, even with mathematicians whose skills they don't have confidence in.
- And soon, they'll understand the foundations of mathematics very differently.

Vladimir Voevodsky, Médaille Fields 2002