Logique et Preuve - CM3

Philippe Duchon – duchon@labri.fr

Année 2019-20

Assistants de preuve

- Le logiciel **Coq** est un *assistant de preuve* : un logiciel qui aide l'utilisateur, par des intéractions avec le logiciel lui-même, à "construire" des preuves formelles d'énoncés.
- Dans le cas de **Coq**, les énoncés sont des *types* dans un certain langage fonctionnel (proche de OCaml), et une preuve d'un énoncé est fournie par n'importe quel *terme* dont le type est l'énoncé (le terme en question est affichable par la commande `Print`).
- **Aujourd'hui :** quelques exemples d'utilisation concrète d'assistants de preuve.
- (et des exemples faciles)

À quoi ça sert ?

- à torturer les étudiants d'informatique ?

À quoi ça sert ?

- à torturer les étudiants d'informatique ? **(oui, mais pas que)**

À quoi ça sert ?

- à torturer les étudiants d'informatique ? **(oui, mais pas que)**
- à produire des preuves mathématiques **plus fiables** quand les preuves dont on dispose sont *extrêmement longues et complexes* (au point de ne pas être vérifiables par un être humain) ; typiquement, des preuves qui se ramènent à vérifier un grand nombre de cas.

À quoi ça sert ?

- à torturer les étudiants d'informatique ? **(oui, mais pas que)**
- à produire des preuves mathématiques **plus fiables** quand les preuves dont on dispose sont *extrêmement longues et complexes* (au point de ne pas être vérifiables par un être humain) ; typiquement, des preuves qui se ramènent à vérifier un grand nombre de cas. (théorème des 4 couleurs ; conjecture de Kepler ; …)

À quoi ça sert ?

- à torturer les étudiants d'informatique ? **(oui, mais pas que)**
- à produire des preuves mathématiques **plus fiables** quand les preuves dont on dispose sont *extrêmement longues et complexes* (au point de ne pas être vérifiables par un être humain) ; typiquement, des preuves qui se ramènent à vérifier un grand nombre de cas. (théorème des 4 couleurs ; conjecture de Kepler ; ...)
- à produire des logiciels **certifiés** (dont on a prouvé formellement qu'ils calculent bien ce qu'on souhaite qu'ils calculent)

À quoi ça sert ?

- à torturer les étudiants d'informatique ? **(oui, mais pas que)**
- à produire des preuves mathématiques **plus fiables** quand les preuves dont on dispose sont *extrêmement longues et complexes* (au point de ne pas être vérifiables par un être humain) ; typiquement, des preuves qui se ramènent à vérifier un grand nombre de cas. (théorème des 4 couleurs ; conjecture de Kepler ; ...)
- à produire des logiciels **certifiés** (dont on a prouvé formellement qu'ils calculent bien ce qu'on souhaite qu'ils calculent)
- (Bien sûr, pour ça il faut savoir définir formellement ce qu'on veut calculer...)

Preuves en mathématiques

- Énoncer et démontrer des théorèmes, c'est en général le travail des mathématiciens.
- Généralement, les preuves sont écrites dans des articles, mais dans un langage qui est beaucoup plus proche du langage naturel (souvent avec des formules, quand même) que des preuves formelles.
- Avant d'être publié dans une revue scientifique, l'article est évalué par un ou plusieurs relecteurs, qui sont censés vérifier que les preuves sont correctes.
- Il arrive que les preuves soient d'une grande complexité, et écrites de manière un peu floue ou rapide ("On ne détaille que le premier cas ; les autres sont similaires") ; ceci est une source potentielle d'erreurs.

Le théorème des 4 couleurs

- Théorème fameux de théorie des graphes : *pour tout graphe qui peut être dessiné dans le plan sans croisement d'arêtes, il est possible de "colorier" les sommets avec 4 couleurs de telle sorte que, pour chaque arête, les deux extrêmités reçoivent des couleurs différentes.*

Le théorème des 4 couleurs

- Théorème fameux de théorie des graphes : *pour tout graphe qui peut être dessiné dans le plan sans croisement d'arêtes, il est possible de "colorier" les sommets avec 4 couleurs de telle sorte que, pour chaque arête, les deux extrêmités reçoivent des couleurs différentes.*
- Conjecturé en 1852, le problème reste ouvert pendant plus d'un siècle.

Le théorème des 4 couleurs

- Théorème fameux de théorie des graphes : *pour tout graphe qui peut être dessiné dans le plan sans croisement d'arêtes, il est possible de "colorier" les sommets avec 4 couleurs de telle sorte que, pour chaque arête, les deux extrêmités reçoivent des couleurs différentes.*
- Conjecturé en 1852, le problème reste ouvert pendant plus d'un siècle.
- **1976** : première preuve (Appel et Haken) : on montre qu'il suffit d'examiner un nombre fini de contre-exemples potentiels (1936), et un programme informatique est écrit pour vérifier que chacun est bien 4-coloriable.

Le théorème des 4 couleurs

- Théorème fameux de théorie des graphes : *pour tout graphe qui peut être dessiné dans le plan sans croisement d'arêtes, il est possible de "colorier" les sommets avec 4 couleurs de telle sorte que, pour chaque arête, les deux extrêmités reçoivent des couleurs différentes.*
- Conjecturé en 1852, le problème reste ouvert pendant plus d'un siècle.
- **1976** : première preuve (Appel et Haken) : on montre qu'il suffit d'examiner un nombre fini de contre-exemples potentiels (1936), et un programme informatique est écrit pour vérifier que chacun est bien 4-coloriable.
- Problème philosophique dans la communauté mathématique : peut-on considérer qu'un théorème est prouvé, si aucun être humain n'a la possibilité de vérifier la preuve ? et si le programme était buggé ?

Le théorème des 4 couleurs

- Théorème fameux de théorie des graphes : *pour tout graphe qui peut être dessiné dans le plan sans croisement d'arêtes, il est possible de "colorier" les sommets avec 4 couleurs de telle sorte que, pour chaque arête, les deux extrêmités reçoivent des couleurs différentes.*
- Conjecturé en 1852, le problème reste ouvert pendant plus d'un siècle.
- **1976** : première preuve (Appel et Haken) : on montre qu'il suffit d'examiner un nombre fini de contre-exemples potentiels (1936), et un programme informatique est écrit pour vérifier que chacun est bien 4-coloriable.
- Problème philosophique dans la communauté mathématique : peut-on considérer qu'un théorème est prouvé, si aucun être humain n'a la possibilité de vérifier la preuve ? et si le programme était buggé ?
- **2005** : preuve en Coq (Gonthier) (la confiance ne repose "plus que" sur le noyau de Coq, qui est un "petit" logiciel, et la formulation des **énoncés**, qui est beaucoup plus facile à vérifier que leurs **preuves**)

La conjecture de Kepler

- Informellement, la *conjecture de Kepler* dit qu'on ne peut pas empiler des sphères (oranges) identiques, de manière plus dense que "ce que font les épiciers" (dans un plan, en réseau hexagonal ; puis chaque niveau suivant dans les "creux" du niveau suivant). Ceci laisse environ 26% de "vide".

La conjecture de Kepler

- Informellement, la *conjecture de Kepler* dit qu'on ne peut pas empiler des sphères (oranges) identiques, de manière plus dense que "ce que font les épiciers" (dans un plan, en réseau hexagonal ; puis chaque niveau suivant dans les "creux" du niveau suivant). Ceci laisse environ 26% de "vide".
- Preuve dûe à **T. Hales (1998)** : 250 pages de notes et 3Go de programmes et de données – invérifiable par des humains.

La conjecture de Kepler

- Informellement, la *conjecture de Kepler* dit qu'on ne peut pas empiler des sphères (oranges) identiques, de manière plus dense que "ce que font les épiciers" (dans un plan, en réseau hexagonal ; puis chaque niveau suivant dans les "creux" du niveau suivant). Ceci laisse environ 26% de "vide".
- Preuve dûe à **T. Hales (1998)** : 250 pages de notes et 3Go de programmes et de données – invérifiable par des humains.
- **2003-2015 :** projet collaboratif pour produire une preuve formelle, utilisant deux assistants de preuve (`Isabelle` et `HOL`).

Le projet CompCert

- **CompCert** est un compilateur C *certifié*, développé par une équipe Inria.

Le projet CompCert

- **CompCert** est un compilateur C *certifié*, développé par une équipe Inria.
- Les compilateurs sont des logiciels horriblement complexes (notamment à cause des optimisations), et les études montrent qu'ils contiennent à peu près systématiquement des bugs.

Le projet CompCert

- **CompCert** est un compilateur C *certifié*, développé par une équipe Inria.
- Les compilateurs sont des logiciels horriblement complexes (notamment à cause des optimisations), et les études montrent qu'ils contiennent à peu près systématiquement des bugs.
- Le développement de **CompCert** s'accompagne d'une **preuve en Coq** de ce que *pour tout programme C compilé par CompCert, le comportement du programme compilé est conforme à ce que décrit la norme ANSI C99* (en d'autres termes, la compilation n'introduit pas d'erreurs qui ne soient pas déjà présentes dans le source).

Le projet CompCert

- **CompCert** est un compilateur C *certifié*, développé par une équipe Inria.
- Les compilateurs sont des logiciels horriblement complexes (notamment à cause des optimisations), et les études montrent qu'ils contiennent à peu près systématiquement des bugs.
- Le développement de **CompCert** s'accompagne d'une **preuve en Coq** de ce que *pour tout programme C compilé par CompCert, le comportement du programme compilé est conforme à ce que décrit la norme ANSI C99* (en d'autres termes, la compilation n'introduit pas d'erreurs qui ne soient pas déjà présentes dans le source).
- Le code compilé produit est moins performant qu'avec d'autres compilateurs classiques ; **CompCert** est prévu pour des programmes *critiques*, typiquement des programmes dont la correction a par ailleurs été prouvée formellement.

Listes en Coq

- Les listes (polymorphes) sont définies en Coq dans un module `List`

```
Inductive list {A: Type} : Type :=
| nil
| cons (a:A)(l:list A).
```

Listes en Coq

- Les listes (polymorphes) sont définies en Coq dans un module `List`

```
Inductive list {A: Type} : Type :=
| nil
| cons (a:A)(l:list A).
```

- Polymorphes : les listes d'éléments de n'importe quel type sont définies d'un seul coup (mais tous les éléments d'une même liste doivent quand même être du même type).

Prouver des choses sur les listes

- Quand on définit le type `list`, Coq définit automatiquement un moyen de prouver des propriétés sur ces listes :

```
Check list_ind.
```

```
list_ind
  : forall (A : Type) (P : list A -> Prop),
    P nil ->
    (forall (a : A) (l : list A), P l -> P (a :: l)) ->
    forall l : list A, P l
```

Prouver des choses sur les listes

- Quand on définit le type `list`, Coq définit automatiquement un moyen de prouver des propriétés sur ces listes :

```
Check list_ind.
```

```
list_ind
  : forall (A : Type) (P : list A -> Prop),
    P nil ->
    (forall (a : A) (l : list A), P l -> P (a :: l)) ->
    forall l : list A, P l
```

- C'est un **principe d'induction** pour les listes !

Retournement de listes

- La fonction `rev` (retournement de liste) est également définie (`::` est un raccourci pour `cons` ; `++` est la concaténation de listes)

```
Fixpoint rev = fun {A : Type} (l : list A) : list A =>
match l with
| nil => nil
| x :: l' => rev l' ++ (x :: nil)
end
```

Retournement de listes

- La fonction `rev` (retournement de liste) est également définie (`::` est un raccourci pour `cons` ; `++` est la concaténation de listes)

```
Fixpoint rev = fun {A : Type} (l : list A) : list A =>
match l with
| nil => nil
| x :: l' => rev l' ++ (x :: nil)
end
```

- (On note que `rev` n'est pas écrite en récursivité terminale)

Listes en Coq (2)

- Le module `List` comprend pas mal de théorèmes sur `rev` :

```
SearchAbout (rev _ = _).
```

```
rev_app_distr:
  forall (A : Type) (x y : list A),
    rev (x ++ y) = rev y ++ rev x
rev_unit:
  forall (A : Type) (l : list A) (a : A),
    rev (l ++ a :: nil) = a :: rev l
```

Prouvons un théorème

- On va prouver un théorème sur la fonction `rev` (qui y est déjà, mais bon) :

```
Theorem rev_involutive:
  forall {A: Type} (l: list A), rev (rev l) = l.
```

Un autre théorème

- Écrivons une version récursive terminale de `rev`

```
Fixpoint revaux {A: Type} (l acc: list A) : list A :=
  match l with
    | nil => acc
    | x::l' => revaux l' (x::acc)
  end.

Fixpoint myrev {A: Type} (l: list A) : list A :=
  revaux l nil.
```

Un autre théorème

- Écrivons une version récursive terminale de `rev`

```
Fixpoint revaux {A: Type} (l acc: list A) : list A :=
  match l with
    | nil => acc
    | x::l' => revaux l' (x::acc)
  end.

Fixpoint myrev {A: Type} (l: list A) : list A :=
  revaux l nil.
```

- On va **prouver** qu'elle calcule toujours la même chose que `rev`

```
Theorem myrev_rev: forall {A: Type} (l: list A),
  myrev l = rev l.
```

Un autre exemple : le tri (par insertion)

- On écrit une fonction d'insertion dans une liste triée

```
Function insert (a:A) (l: list A) : list A:=
  match l with
      [] => [a]
    | b::l' => if  a <= b then a::l else b::insert a l'
  end.
```

Un autre exemple : le tri (par insertion)

- On écrit une fonction d'insertion dans une liste triée

```
Function insert (a:A) (l: list A) : list A:=
  match l with
      [] => [a]
    | b::l' => if  a <= b then a::l else b::insert a l'
  end.
```

- On en tire une fonction de tri par insertion

```
Function sort (l: list A) : list A :=
  match l with
      nil => nil
    | a::l' => insert a (sort l')
  end.
```

- On définit deux prédicats : “être trié” et “contenir les mêmes éléments”

```
Inductive Sorted : list A -> Prop :=
  Sorted_0 : Sorted nil
 | Sorted_1 : forall a, Sorted [a]
 | Sorted_2 : forall a b l, a <= b  ->
                        Sorted (b::l) ->
                        Sorted (a::b::l).
Inductive Permutation :
      list A -> list A -> Prop :=
   perm_nil : Permutation [] []
 | perm_skip : forall (x : A) (l l' : list A),
               Permutation l l' -> Permutation (x :: l) (
 | perm_swap : forall (x y : A) (l : list A),
               Permutation (y :: x :: l) (x :: y :: l)
 | perm_trans : forall l l' l'' : list A,
                Permutation l l' -> Permutation l' l''
                    -> Permutation l l''
```

- On définit le prédicat “être une fonction de tri”

```
Definition Sort_spec (f : list A -> list A) :=
  forall l, let l' := f l in
       Permutation l' l /\ Sorted l'.
```

- On définit le prédicat “être une fonction de tri”

```
Definition Sort_spec (f : list A -> list A) :=
  forall l, let l' := f l in
       Permutation l' l /\ Sorted l'.
```

- On prouve alors un théorème de correction :

```
Theorem sort_correct : Sort_spec sort.
```

- On définit le prédicat “être une fonction de tri”

```
Definition Sort_spec (f : list A -> list A) :=
  forall l, let l' := f l in
       Permutation l' l /\ Sorted l'.
```

- On prouve alors un théorème de correction :

```
Theorem sort_correct : Sort_spec sort.
```

- (En fait il y a quelques lemmes intermédiaires, voir fichier `sort.v`)

- On définit le prédicat “être une fonction de tri”

```
Definition Sort_spec (f : list A -> list A) :=
  forall l, let l' := f l in
       Permutation l' l /\ Sorted l'.
```

- On prouve alors un théorème de correction :

```
Theorem sort_correct : Sort_spec sort.
```

- (En fait il y a quelques lemmes intermédiaires, voir fichier `sort.v`)
- On peut même extraire une traduction en OCaml

```
Extraction Language Ocaml.
Extraction "mysort.ml" sort .
```