Introduction à Rocq pour les CPGE

Yves Bertot

Avril 2025

Ce que Rocq apporte

- Un interprète de programmation fonctionnelle pure (proche d'OCaml)
- La possibilité de s'exercer à raisonner sur du code
- Une utilisation directe dans les browser web
- `https://rocq-prover.github.io/platform-docs/Tutorial_Equations_basics.html`
- `https://rocq-prover.org/install`

Quels rapports avec l'informatique en CPGE

- Programmation en Coq proche de OCaml
- Quelques points tirés des programmes d'informatique (MP2I et MPI)
  - décrire et spécifier, démontrer un algorithme par une preuve mathématique
  - sensibiliser à une validation formelle
  - Terminaison, Correction partielle, etc.

Le parti-pris de cette présentation

- Utiliser un paquet “avancé” de définition de fonctions
- Profiter d'une approche puissante pour raisonner par récurrence
- Fournir aux étudiants des exercices et outils probables
- Programmes impératifs, avec boucles et invariants
- Réfléchir aux difficultés potentielles pour l'étudiant et le professeur

L'outil Equations

```
From Stdlib Require Import ZArith Arith List Lia Bool.
From Stdlib Require Import Program Zwf.
From Equations Require Import Equations.
From Equations.Prop Require Import Logic.

Equations tail {A} (l : list A) : list A :=
  tail nil := nil;
  tail (a :: v) := v.

Equations last {A} (a : A) (l : list A) : A :=
  last a nil := a;
  last a (b :: tl) := last b tl.
```

Remarques sur `tail` et `last`

- Utilisation d'arguments implicites : `{A}`
  - Similaire au typage polymorphe d'Ocaml
- Notation `::` similaire à OCaml
- Définition par règles de filtrage comme dans `match`
  - `:=` remplace `->`, `;` remplace `|`
- Obligation de couvrir tous les cas du type de données
  - Pas d'exception, pas d'erreur à run-time
- `last` est une fonction récursive terminante
- Une distinction notable vis-à-vis d'Ocaml

Obligation de termination

- L'outil vérifie que la fonction définie terminera toujours
- Ceci impose une discipline au programmeur
  - Inconvénient: connaissances supplémentaires à acquérir
  - On peut éviter cet effort en admettant le résultat

Exemple de définition sans preuve de terminaison

```
Equations Zfact1 (x : Z) : Z by wf x (Zwf 0) :=
  Zfact1 x with (x <=? 0)%Z := {
    Zfact1 x true := 1;
    Zfact1 x false := x * Zfact1 (x - 1)
  }.
Next Obligation. admit. Admitted.
```

- L'argument de décroissance est mentionné `wf x (Zwf 0)`
- La preuve de décroissance est admise

Laisser entrer le loup dans la bergerie

```
Equations Zfact_bad (x : Z) : Z by wf x (Zwf 0) :=
  Zfact_bad x with (x =? 0)%Z := {
    Zfact_bad x true:= 1;
    Zfact_bad x false := x * Zfact_bad (x - 1)
  }.
Next Obligation. admit. Admitted.
```

- Admettre que `Zfact_bad` est bien définie est incohérent
  - Les garanties fournies par la preuve formelle disparaissent
- Rocq s'en protège en gardant trace des hypothèses utilisées
- Peut-être pas gênant dans un contexte pédagogique

Avec preuve de terminaison

```
Equations Zfact (x : Z) : Z by wf (Z.abs_nat x) lt :=
  Zfact x with inspect (x <=? 0) := {
    Zfact x (true eqn:p) := 1;
    Zfact x (false eqn:p) := x * Zfact (x - 1)
  }.
Next Obligation.
lia.
Qed.
```

- Besoin d'ajouter la clause `inspect`
- `p` est l'information que le test a échoué
  utile pour prouver la terminaison

Faire des preuves

- Ecrire des énoncés universels
- Quelques prédicats définis : égalité, comparaison de nombres
- Compléter avec des fonctions ou des calculs de test
- Utiliser aussi des quantifications existentielles

Un point de vue pragmatique sur les preuves

- Très similaire au test en boite blanche
  - On ne garantit que ce que l'on prouve
  - Par exemple: plusieurs raisons pour un tri d'être incorrect: résultat non trié, perte ou ajout de données, instabilité
- Meilleure couverture
  - La preuve couvre tous les cas d'exécution
- Calcul symbolique sur les variables de l'énoncé
- La preuve par récurrence est le miroir de la programmation récursive

Exemple de propriété à prouver sur la factorielle

```
Lemma Zfact_multiple (x y : Z) :
  0 < y <= x -> exists k, Zfact x = k * y.
```

Preuves par récurrence

- Preuve par récurrence sur les nombres naturels
  - Connue par les étudiants (cours de maths)
- Preuve par récurrence structurelle sur les données
  - Approche traditionnelle de l'enseignement Coq/Rocq
  - similaire (mais plus faible) qu'une preuve sur la taille des arguments
- Preuve par récurrence sur les fonctions récursives
  - Lié à `Equations`
  - Travaux précurseurs de Barthe et al. (2006)

Récurrence sur les nombres naturels

- Toute propriété vraie pour 0 et préservée par passage au successeur est vraie pour tous les entiers naturels
- En pratique: couvrir un ensemble infini en prouvant seulement 2 cas
- Existe aussi avec une variante forte
- Interprétation informatique
  - Le cas de base produit une preuve de la propriété pour 0
  - Le cas récursif produit une preuve de la propriété pour 1, puis pour 2
  - Il suffit de répéter

Récurrence structurelle

- Pour tous les types d'arbres
- Toute propriété satisfaite pour les nœuds atomiques et préservée par construction par les autres nœuds est vraie pour tous les arbres du type
- En Rocq, le principe de récurrence est engendré automatiquement pour chaque définition de type d'arbre
- La récurrence sur les nombres naturels est un cas particulier

Exemple pour les arbres binaires

```
Inductive btree {A} : Type :=
  L (x : A) | N (t1 t2 : btree).
```

Pour un type `A` fixé, toute propriété satisfaite pour tout arbre de la forme `L a`, quelque soit `a` dans `A`, et vraie pour `N t1 t2` dès qu'elle est déjà satisfaite pour `t1` et `t2` est satisfaite pour tous les arbres.

- Notez l'utilisation de deux hypothèses de récurrence

Exemple pour les listes

Pour un type `A` fixé, Toute propriété vraie pour la liste vide, et vraie pour toute liste de la forme `a :: l` avec a dans `A` dès qu'elle est vraie pour `l` est vraie pour toutes les listes à eléments dans `A`.

Récurrence sur les fonctions

On s'intéresse aux propriétés à plusieurs arguments reliant les arguments d'une fonction et le résultat de cette fonction

- Exemple déjà vu: le résultat de la fonction factorielle d'un nombre positif est divisible par tous les prédécesseurs de l'argument

Toute propriété satisfaite pour les arguments et le résultat dans les cas de base, et satisfaite dans les cas avec appels récursif dès qu'elle était déjà satisfaite pour les appels récursif est satisfaite pour toutes les exécutions qui terminent

Analogie avec la récursion structurelle

- Toute exécution d'une fonction récursive fait apparaitre un arbre d'appel
  - Les nœuds internes sont les appels récursifs
- Chaque chemin d'exécution dans la fonction correspond à un cas
- Pour couvrir tous les arbres d'appels il faut couvrir tous les cas d'exécution

Exemple factorielle

```
Equations Zfact (x : Z) : Z by wf (Z.abs_nat x) lt :=
  Zfact x with inspect (x <=? 0):= {
    Zfact x (true eqn:p) := 1;
    Zfact x (false eqn:p) := x * Zfact (x - 1)
  }.
```

- Un chemin d'exécution par équation (ici)
- Un chemin sans appel récursif
- Un chemin avec un appel récursif
- Preuves par récurrence sur cet arbre similaires aux preuves sur les listes

Avec Rocq

- On démarre une preuve en exprimant un énoncé logique (le but)
- On décompose en sous-but (qui suffisent)
- Dans chaque but on a deux types d'information
  - Les faits connus
  - La question à résoudre

Tactiques

- Commandes pour décomposer les buts
- Manipulation des connecteurs logiques
- Preuves par récurrence
- Preuve automatique
  - `lia` : arithmétique entière linéaire
  - `ring` : égalités entre formules polynomiales
- `https://www-sop.inria.fr/members/Yves.Bertot/courses/2025/cheatsheet.pdf`

Tactiques automatiques : `lia`

- Le but doit être une comparaison entre des formules linéaires
- Le contexte peut contenir des comparaisons entre formules linéaires
- Répond favorablement quand le but est une conséquence du contexte

```
Require Import ZArith Lia.
Open Scope Z_scope.

Lemma lia_example: forall x y,
  2 * x + 1 < y -> 3 * y + 1 < x -> x < 0.
Proof.
lia.
Qed.
```

Tactiques automatiques : `ring`

- Egalités entre formules polynomiales
- Pas d'utilisation du contexte

```
Require Import ZArith.

Lemma ring_example : forall x y,
  (x + y) ^ 2 = x ^ 2 + 2 * x * y + y ^ 2.
Proof.
intros x y.
ring.
Qed.
```

Trouver et utiliser des théorèmes

- `Search` *pattern* . . . *pattern*
- Les théorèmes dont la conclusion est une égalité s'utilisent avec `rewrite`
- Les autres s'utilisent avec `apply`

Structurer par des étapes intermédiaires

- `assert (un_nom : énoncé).` Permet de rendre le script plus lisible.
- `enough (un_nom : énoncé) by easy.`
- Le professeur peut ajouter des phrases types et recommander leur utilisation.
- Voir aussi Waterproof Lean-verbose pour les preuves déclaratives en math

Démonstration: la propriété de factorielle

- Ça a l'air évident
- Mais pas pour le sytème de preuve
- Je vais montrer deux variantes, une opérationnelle et une déclarative

Exemple : `merge` et merge sort

- `merge` est difficile pour la récursion structurelle
- Plus facile d'utiliser une mesure
  - la somme des longueur des arguments
- Pour implémenter `merge_sort` il faut aussi couper l'entrée en morceaux
- There are other variants using lists of lists (Okasaki 1996)
  - version standard de la librairie de Rocq
- Utilisation d'arbres binaires

Raisonner sur des programmes impératifs avec boucles

- Pas le meilleur outil
- Utilisation de la programmtion récursive terminale
- Utiliser un état et une borne sur le nombre maximal d'itération
- Raisonner à l'aide d'invariants

La construction `while_loop`

```
Equations while_loop {A} (n : nat) (test : A -> bool)
  (body : A -> A) (a : A) : A + A :=
  while_loop 0 test body a with inspect (test a) := {
    | false eqn: h => inl a
    | true eqn: h => inr a};
  while_loop (S p) test body a with inspect (test a) := {
    | false eqn: h => inl a
    | true eqn: h => while_loop p test body (body a)}.
```

- Sortie avec `inl` dès que le test renvoie `false`
- Sortie avec `inr` si la borne est dépassée
- Le type `A` doit être choisi pour représenter l'état qui est modifié à chaque itération

Raisonnement par invariant, correction partielle

Pour une propriété `P` sur les états (de type `A`)

- Préservée par exécution du corps de la boucle
  - En supposant que le test est satisfait initialement
- Satisfaite pour l'état initial

Si la boucle termine, cette propriété est satisfaite pour l'état final

- Et en plus, le test est faux sur l'état final

Exemple de boucle

```
x = n;
y = 0;
while (0 < x) {
 x = x - 1;
 y = y + x
}
```

Encodage en Rocq

```
Record state := {x : Z; y : Z}.

Definition sample_start n := {| x := n; y := 0 |}.

Definition sample_test (s : state) := 0 <? x s.

Definition sample_body (s : state) :=
  let s1 := {| x := x s - 1; y := y s|} in
  {| x := x s1; y := y s1 + x s1 |}.

Definition sample_program (n : Z) :=
  while_loop (Z.abs_nat (2 * n))
    sample_test sample_body (sample_start n).
```

Proof of the sum of the first integers

```
Lemma program_proof n s :
    0 <= n ->
    sample_program n = inl s -> y s = n * (n - 1) / 2.
```

Mise en place

Je conseille d'installer la plateforme Coq, mais j'ai peu d'expérience
Sur ma machine (ubuntu 22.04.5 LTS), je pense avoir installé `opam` puis utilisé `opam` pour installer le reste

```
sudo apt-get install opam
opam repo add coq-released \
  https://coq.inria.fr/opam/released

opam pin add rocq-prover 9.0.0
opam install rocq-equations
```

Si vous utilisez une installation coq-platform (sur Mac ou Windows), alors vous aurez peut-être une installation avec coq 8.20.1 ou 8.19
Je fournis deux fichiers d'exemples. Le fichier `test_equations.v` founction avec `rocq-prover` 9.0.0 Le fichier `test_equations_v8_20.v` a été test avec coq 8.20.1 et marche probablement avec une version plus ancienne