Partie 2 : Introduction à la preuve formelle avec Coq

François Thiré

February 1, 2019

1 Introduction

Ce projet est découpé en une série de petits projets. À vous de choisir ceux qui vous intéressent puis ensuite de les formaliser en Coq avec SSreflect. Je vous invite à essayer d'aller le plus loin possible dans vos idées quitte à laisser de temps en temps des preuves incomplètes ou bien des définitions pas tout à fait correctes puisqu'une partie de l'évaluation sera consacrée à vos choix de modélisation de ces problèmes. Les étoiles sont une indication de la difficulté estimée des questions. Vous trouverez ci-dessous le nombre de questions par niveau de difficulté :

2 Trier une liste

Dans la suite de ce qu'on a fait au TP précédent, je vous demande de programmer une fonction $sort : \mathbb{L}_{\mathbb{N}} \to \mathbb{L}_{\mathbb{N}}$[1] qui trie une liste d'entiers et de la certifier correcte.

Tâches à effectuer :

/ Choisir un algorithme de tri et implémenter la fonction *sort* en utilisant cet algorithme

* Définir un prédicat *sorted* qui certifie qu'une liste est triée

* Définir un prédicat binaire *permuted* qui certifie qu'une liste est la permutation d'une autre

*** Prouver que votre algorithme de tri est correct vis-à-vis de ces deux prédicats

* Est-il possible d'étendre votre algorithme de tri pour n'importe quel type ? Quelles sont les hypothèses *minimales* nécessaires ? Sous ces hypothèses étendez votre développement pour prendre en compte ces nouveaux types

[1] $\mathbb{L}_{\mathbb{X}}$ désigne l'ensemble des listes dont les éléments appartiennent au type X

** Définir l'algorithme du *quicksort* en Coq.

*** Prouver la correction de cet algorithme

3 Les entiers binaires

Au lieu de représenter nos entiers à l'aide des constructeurs `O` et `S` de façon unaire, on pourrait utiliser une représentation binaire avec trois constructeurs en utilisant le type inductif suivant :

```
  Inductive bin : Set :=
| Zero : bin
| Double : bin -> bin
| DoubleOne : bin -> bin.
```

où moralement le constructeur `Double` double son entrée et `DoubleOne` double son entrée et ajoute un. Le type de ces nouveaux entiers sera noté $\mathbb{N}_2$.

Tâches à effectuer :

* Programmer deux fonctions $f : \mathbb{N} \to \mathbb{N}_2$ et $g : \mathbb{N}_2 \to \mathbb{N}$ telles que $g \circ f = id_{\mathbb{N}}$

* Prouver cette dernière égalité

* Est-ce vrai que $f \circ g = id_{\mathbb{N}_2}$? Si ce n'est pas le cas, programmer une fonction $h : \mathbb{N}_2 \to \mathbb{N}_2$ telle que $\forall x, g(x) = g(h(x))$ et $f \circ g \circ h = h$. Il vous ait demandé de créer une autre fonction h que $f \circ g$.

*** Prouver ces deux dernières égalités. La difficulté de cette question dépend grandement de votre définition pour h.

** Pouvez-vous trouver une autre représentation des entiers binaires qui a une représentation unique pour chaque entier et donc prouver son isomorphisme facilement ?

4 Rajouter la logique classique en Coq

Coq utilise une logique constructive. Cela implique par exemple que le tiers-exclu n'est pas prouvable. Il est possible d'étendre la logique de Coq en précédant la déclaration du nouvel axiome par le mot-clé `Axiom`. Une façon de rajouter la logique classique est d'ajouter l'axiome du tiers-exclu, mais ce n'est pas la seule solution possible. On peut aussi vouloir rajouter l'élimination de la double-négation par exemple. Voici 6 façons d'introduire la logique classique en Coq :

- La loi de Peirce

  ```
  forall (P Q:Prop), ((P -> Q) -> P) -> P.
  ```

- L'élimination de la double négation

  ```
  forall (P:Prop), ~~P -> P.
  ```

- Le tiers-exclu

  ```
  forall (P:Prop), P \/ ~P
  ```

- La définition classique (au sens logique) de l'implication

  ```
  forall (P Q:Prop), (P -> Q) -> ~P \/ Q
  ```

- Une version modifiée de la loi de De Morgan

  ```
  forall (P Q:Prop), ~(~P /\ ~Q) -> P \/ Q
  ```

- Une version modifiée de la double négation

  ```
  forall (P:Prop), (~P -> P) -> P
  ```

Tâches à effectuer :

*** Montrer constructivement (sans nouvel axiome) l'équivalence entre tous ces axiomes.

** Parmi les loi 4 implications de De Morgan, 3 sont prouvables en Coq, lesquelles ?

** Comment prouver que cette dernière loi n'est pas prouvable en Coq ?

** De la même manière, comment prouver que cette loi n'est pas équivalente à l'une des lois ci-dessus ?

** Pouvez-vous trouver d'autres axiomes non-triviaux qui ont le même statut que cette dernière loi de De Morgan ?

5 Le principe des tiroirs

Il existe plusieurs façons de modéliser le théorème des tiroirs en Coq. Une façon est d'utiliser les listes : soit X un type arbitraire et deux listes $l_1, l_2 \in \mathbb{L}_X$, alors si $\forall x, x \in l_1 \Rightarrow x \in l_2$ et que $|l_2| < |l_1|$ alors la liste l_1 *se répète*.

Tâches à effectuer :

* Définir un prédicat *simple* `repeats` qui est vrai si et seulement si la liste l_1 *se répète*. En particulier, on ne prendra pas comme prédicat la prémisse du théorème.

* Modéliser l'énoncé ci-dessus en Coq

*** Prouver votre énoncé (vous pouvez utiliser un des axiomes classiques présentés ci-dessus.

Il se trouve qu'il est aussi possible de prouver cet énoncé constructivement, cependant cela demande une astuce : avec les hypothèses du principe des tiroirs, il est possible de plonger notre liste l_1 dans une liste l_3 de type $\mathbb{L}_{\mathbb{N}}$ tel que l'énoncé `repeats l3 -> repeats l1` soit prouvable.

Tâches à effectuer :

**** Prouver votre énoncé du principe des tiroirs constructivement.

6 Well-foudness en Coq

En Coq, il existe un prédicat `well_founded` dont l'implémentation est comme suit

```
Definition well_founded (A : Type) (R : A -> A -> Prop) : Prop =
  forall a : A, Acc R a.
```

avec la définition suivante de `Acc` :

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

L'intuition derrière `Acc`, est de dire que `Acc R x` est prouvable (autrement dit `x` est accessible) si tous les éléments plus petits que `x` sont eux-mêmes accessibles selon la relation `R`. Le cas de base étant quand `x` est un élément minimal.

Tâches à effectuer :

* Prouver que l'ordre naturel sur les entiers est bien fondé

** Prouver que l'ordre produit sur les couples d'entiers naturels est bien fondé

** Prouver que l'ordre lexicographique sur les couples d'entiers naturels est bien fondé

*** Prouver que l'ordre lexicographique sur les listes d'entiers naturels décroissant est bien fondé

*** Définir un ordre $<_{div}$ pour la relation de divisibilité. Prouver sa bonne fondaison.

7 Les ordinaux de Brouwer

Il est possible de modéliser en Coq les ordinaux[2] en utilisant le type inductif suivant :

```
  Inductive Ord : Set :=
| Zero : Ord
| Succ : Ord -> Ord
| Limit : (nat -> Ord) -> Ord.
```

[2] noté $\mathbb{O}$

Tâches à effectuer :

* Définir l'ordinal ω^3 en utilisant le type des ordinaux présenté ci-dessus

*** Avec cette définition des ordinaux, est-ce vrai qu'ω est le plus petit ordinal limite ? Dans le cas contraire, rajoutez des hypothèses non triviales et montrer qu'ω est le plus petit ordinal limite sous ces hypothèses

** Définir la fonction $+_{\mathbb{O}}$ sur les ordinaux (décroissante sur le second argument et non-commutative)

* Est-il possible de prouver que $2 +_{\mathbb{O}} \omega = \omega$?

** Définir une relation d'ordre $\leq_{\mathbb{O}}$ sur les ordinaux, et prouvez que c'est une relation d'ordre

* En utilisant $\leq_{\mathbb{O}}$, en déduire une relation d'équivalence $\equiv_{\mathbb{O}}$ sur les ordinaux, et prouver que c'est bien une relation d'équivalence

*** En déduire que $2 + \omega \equiv_{\mathbb{O}} \omega$

** Donner une définition de l'ordinal ϵ_0[4]

8 Les formes normales de Cantor

À la section précédente, on s'est aperçu qu'il n'est pas forcément évident de travailler avec les ordinaux de Brouwer en Coq, notamment car le constructeur `Limit` manipule des fonctions. Pour manipuler des ordinaux jusqu'à ϵ_0 il est possible d'utiliser la notion de *forme normale de Cantor*. L'idée est que – de façon analogue aux entiers naturels – on peut représenter nos ordinaux dans une base. Pour les ordinaux inférieurs à ϵ_0, on peut prendre la base ω. La forme normale de Cantor est une façon canonique de représenter un ordinal sur cette base. Soit α un ordinal[5], alors on écrit

$$\alpha = \omega^{\beta_0} c_0 + \cdots + \omega^{\beta_k} c_k$$

avec $c_i \in \mathbb{N}^*$ et β_i une suite d'ordinaux strictement décroissants. Il est aussi possible de définir ces formes normales avec tous les $c_i = 1$ et en autorisant à ce que les β_i soient seulement décroissants.

Tâches à effectuer :

** Proposer une modélisation des ordinaux en forme normale de Cantor en Coq, cela vous demandera de définir aussi une relation d'ordre $<_{\mathbb{O}}$.

*** Implémenter une fonction $+_{\mathbb{O}}$ sur ces ordinaux et montrer que le résultat est en forme normale de Cantor (cela peut-être trivial en fonction de votre modélisation)

[3] le plus petit ordinal limite
[4] le plus petit point fixe de la fonction $f(\alpha) = \omega^\alpha$
[5] qui ne soit pas le plus petit ordinal

*** Prouver cette fois que $+_{\mathbb{O}}$ est commutative

**** Prouver la bonne fondaison de $<_{\mathbb{O}}$

9 Le problème de l'Hydre

Le jeu de l'Hydre se base sur la mythologie grecque où Hercule devait tuer une Hydre. Ce monstre mythologique qui avait plusieurs têtes avait la particularité que si on lui coupait une tête, deux autres têtes repoussaient. Les informaticiens Paris et Kirby ont transformé ce conte mythologique pour en faire un jeu[6].

Dans ce jeu, Hercule et l'Hydre jouent tour à tour et Hercule cherche à couper toutes les têtes de l'Hydre. À chaque tour, Hercule choisit de couper une tête, tandis que l'Hydre peut se faire régénérer autant de fois qu'elle le souhaite des têtes. Le résultat surprenant de ce jeu, est que peu importe la stratégie d'Hercule, il existe toujours une façon pour ce dernier de vaincre l'Hydre. Cela dépend bien sûr du mécanisme de régénération des têtes de l'Hydre qui n'est pas anodin.

Si on cherche à formaliser ce jeu, on peut voir l'Hydre comme un arbre. Les têtes seraient les feuilles de l'arbre, son corps serait la racine, ses cous seraient les branches de l'arbre et enfin les jonctions entre ses cous seraient les noeuds intermédiaires.

Au tour d'Hercule, ce dernier choisit de supprimer une tête ce qui entraine la suppression de la feuille correspondante et de sa branche sous-jacente.

Se présentent alors deux cas. Ou bien le noeud sous-jacent à la feuille est le corps de l'Hydre. Dans ce cas-là, aucune tête ne peut repousser. Sinon, l'Hydre générera à nouveau des têtes en partant du noeud situé en-dessous du noeud sous-jacent à la branche supprimée, en générant autant de copies qu'elle le souhaite de l'arbre situé au-dessus de ce noeud. Pour un peu mieux comprendre comment ce jeu fonctionne, vous pouvez tester le jeu ici : `http://www.crypto.ethz.ch/teaching/lectures/DM15/Hydra-exercise/Hydra.htm`.

Tâches à effectuer :

** Formaliser un nouveau type `Hydra` qui formalisera la notion d'arbre représentant l'Hydre

**** Formaliser une relation `round : Hydra -> Hydra -> Prop` telle que `round h h'` est prouvable si et seulement s'il est possible d'obtenir `h'` à partir de `h` après qu'Hercule a coupé une tête et que l'Hydre s'est régénérée

Pour n'importe quel ordre bien fondé $<$ sur un ensemble $\mathbb{E}$, on dit qu'une fonction de type $\mathbb{E} \to \mathbb{O}$ est *adéquate* si elle est compatible avec l'ordre[7] :

$$\forall x\ y \in \mathbb{E}, x < y \Rightarrow f(x) <_{\mathbb{O}} f(y)$$

[6] on déplore cependant qu'il n'y ait pas de championnat pour ce jeu

[7] on supposera qu'il en existe toujours au moins une

En particulier il existe une plus petite fonction adéquate $f_{\rightarrow}$ au sens où[8]

$$\sup_{x \in \mathbb{E}} f_{\rightarrow}(x) \leq \sup_{x \in \mathbb{E}} f(x)$$

On appelle *mesure* de l'ordre, l'ordinal

$$\sup_{x \in \mathbb{E}} f_{\rightarrow}(x)$$

.

Paris et Kirby ont montré que dans le cadre du jeu de l'Hydre,

$$\sup_{x \in \mathbb{E}} f_{\rightarrow}(x) \ = \epsilon_0$$

.

Tâches à effectuer :

** En utilisant votre représentation favorite des ordinaux, implémenter une fonction adéquate f de type `Hydra -> Ordinal` qui soit compatible avec `round`

**** Montrer que votre fonction f est adéquate

** Montrer qu'il n'existe pas de fonction f compatible avec `round` tel que $\sup f = \omega$

*** Montrer qu'il n'existe pas de fonction f compatible avec `round` tel que $\sup f = \omega^2$

***** Montrer qu'il n'existe pas de fonction f compatible avec `round` tel que $\sup f = \omega^\omega$

10 L'arbre de Stern-Brocot

L'arbre de Stern-Brocot donne une énumération élégante des rationnels strictement positifs. Il est défini (co)-inductivement de la façon suivante :

- $SB(i,j) = (\frac{i}{j}, SB(i, i+j), SB(i+j, j)$

On obtient une énumeration des rationnels en partant de $SB(1,1)$.
Tâches à effectuer

** Modéliser l'arbre de Stern-Brocot en Coq. À vous de choisir si vous souhaitez utiliser un inductif ou bien un coinductif.

*** Montrer que chaque rationnel dans l'arbre est irréductible

*** Montrer que chaque fraction apparaît au plus une fois dans l'arbre

**** Montrer que n'importe quelle fraction irréductible apparaît dans l'arbre.

[8] *sup* désigne la borne supérieure

11 Modalités d'évluation

Pour ce projet, je vous demande de me fournir deux fichiers. Un fichier `.v` contenant vos preuves et commentaires, et un fichier `.html` qui aura été généré par l'outil `coqdoc`. Je vous demande d'apprendre rapidement comment fonctionne `coqdoc` pour que dans le fichier `html` je puisse y trouver vos commentaires dans un rendu propre. Commenter un fichier coq est difficile ! Le but de ces commentaires étant d'éviter de dérouler les tactiques étapes par étape pour comprendre l'idée principale de votre preuve. Attention non plus à ne pas faire trop de commentaires et à me détailler chaque étape de calcul. N'hésitez-pas à faire relire votre fichier par un de vos condisciples pour avoir un retour pertinent.

Votre rendu devra se présenter sous une archive de la forme `NOM_PRENOM.tar.gz` qui se dézippera en un dossier `NOM_PRENOM` qui contiendra vos deux fichiers `NOM_PRENOM.v` et `NOM_PRENOM.html`.