IN 101 - TD 06

Énoncé

Ce TD est conçu pour être conduit main dans la main avec votre enseignant. Il vous amène à découvrir un outil d'aide à la preuve, avec sa syntaxe peu habituelle. Les constructions nécessaires pour se servir de cet outil vous seront introduites au fur et à mesure.
L'idée de ce TD n'est pas d'être expert avec un tel outil, mais de comprendre par quels mécanismes généraux il est possible de démontrer formellement des propriétés sur des programmes.

1 Introduction

Prouver des propriétés sur un programme peut se faire sur papier. Néanmoins cette pratique ne contraint pas structurellement à une rigueur sans faille. *A contrario*, utiliser un outil d'aide à la preuve apporte au moins deux avantages :

- cela oblige à une rigueur sans faille qui, en échange, garantit qu'une preuve acceptée par l'outil est assurément correcte,
- cela apporte (parfois) une aide durant la recherche et la rédaction de la preuve par l'intermédiaire de :
  - tactiques d'automatisation,
  - procédures automatiques,
  - simplement le fait que l'outil détermine automatiquement les prochains buts à prouver en fonction des commandes de preuves invoquées par l'utilisateur.

Le majeur désavantage de cet exercice est une autre façon de voir le premier avantage énoncé : cela demande parfois un travail réellement conséquent.

On pourra citer comme désavantage secondaire le fait que la preuve obtenue n'est pas forcément très compréhensible à la relecture. Pour autant, est-il vraiment facile de se convaincre intimement de la correction d'une preuve faite à la main, en langue naturelle, avec les nombreux sous-entendus qu'elle comporte ?

2 Environnement

L'outil que nous allons utiliser est Coq (`https://coq.inria.fr`) qui est développé depuis des décennies à l'INRIA.

Plutôt que l'utiliser dans un terminal, nous allons passer par une interface plus sympathique permettant la mise en couleur de la syntaxe et l'évaluation des commandes, avec la visualisation de leurs effets sur l'état courant d'une preuve.

Vous travaillerez sur votre machine sans vous connecter à `info1.ensta.fr`. Conformément à ce qui vous a été préalablement demandé par mail, vous devriez avoir déjà installé ladite interface nommée `CoqIDE`.

Pour rappel, la page de téléchargement se trouve à `https://github.com/coq/platform/releases/tag/2022.09.1`. Sélectionnez la version correspondant à votre machine / système d'exploitation. Sous Linux, l'installation peut également être faite en invoquant la commande `sudo apt install coqide`.

En cas d'impossibilité d'installer Coq sur votre machine, il est possible de tester Coq dans un navigateur en se rendant à l'adresse `https://jscoq.github.io/scratchpad.html`.

Q 2.1. Lancez `CoqIDE` et créez un nouveau fichier que vous sauverez sous le nom `mult.v`. Le suffixe `.v` est celui des « programmes » Coq.

3 C'est parti...

Nous allons travailler sur la fonction *mult* vue en cours dont l'algorithme est le suivant :

```
mult (x, y) =
  si y = 0 alors retourner 0
  sinon retourner x + mult (x, (y - 1))
```

pour laquelle nous voulons prouver un théorème de correction statuant que :

$$P : \forall x, y \in \mathbb{N},\ mult(x, y) = x \times y$$

Q 3.1. Revoyez avec votre enseignant la preuve qui suit (donnée en cours) et assurez-vous de bien la comprendre :

- Soit x un entier naturel, soit y un entier naturel...
- Fonction récursive $\Rightarrow$ preuve par récurrence sur y.
- Deux cas possibles.

- Cas $y = 0$ (1)
  Prouvons que $mult(x, 0) = x \times 0$.
  - Par définition de *mult*, on a $mult(x, 0) = 0$ (2)
  - Nous savons que $\forall n \in \mathbb{N},\ 0 = n \times 0$ donc $0 = x \times 0$. (3)
  - CQFD par (2), (3).
- Cas $y > 0$ (1)
  Prouvons que $mult(x, y) = x \times y$.
  - Par définition de *mult*, on a $mult(x, y) = x + mult(x, (y - 1))$ (2)
    Donc nous devons prouver $x + mult(x, (y - 1)) = x \times y$ (3)
  - Par (1), $y - 1 < y$, donc hypothèse de récurrence applicable. (4)
  - Par hypothèse de récurrence, nous avons $\forall i < y,\ mult(x, i) = x \times i$. (5)
  - Par (4), (5) nous avons $mult(x, (y - 1)) = x \times (y - 1)$. (6)
  - Par (3), (6) nous devons prouver $x + x \times (y - 1) = x \times y$.
  - Donc que $x \times (1 + y - 1) = x \times y$.
  - CQFD par les propriétés de $+$ et $*$.

□

Q 3.2. Écrivez dans votre fichier le « programme » Coq implantant la fonction `mult`. Le code est le suivant (le copier-coller doit fonctionner) et il vous sera détaillé par votre enseignant. Ne soyez pas effrayé par la syntaxe, elle nous importe peu !

```
Require Import ZArith.

Fixpoint mult (x : nat) (y : nat) :=
  match y with
  | O => O
  | S n => x + (mult x n)
  end.
```

Q 3.3. Maintenant que votre programme contenant la fonction `mult` est écrit en Coq, il est temps de le faire compiler par Coq. Pour ce faire, sous `CoqIDE`, positionnez le curseur à la fin de la fonction et sélectionnez l'entrée de menu Navigation ⟩ Run to cursor (un raccourci clavier est disponible et indiqué à côté de cette entrée). Un des boutons de la barre d'outil permet également cette action. Si vous travaillez avec Coq dans un navigateur, utilisez Alt + ↑ et Alt + ↓ pour vous déplacer à une ligne en la faisant interpréter par Coq. Si vous travaillez directement dans un terminal, l'appui sur ⏎ fera automatiqment exécuter la commande.

Q 3.4. Il est maintenant temps d'annoncer le théorème que nous voulons prouver et qui décrit que notre fonction `mult` fait bien son travail : son résultat est celui que rend la multiplication « des maths ». Dans votre fichier, rajoutez la ligne suivante :

```
Lemma mult_correct : forall x y : nat, (mult x y) = (x * y).
Proof.
```

Q 3.5. Il est maintenant temps de commencer à faire notre démonstration. **Nous** allons écrire la preuve : ce n'est pas Coq qui va la faire à notre place. Coq va seulement :

- vérifier qu'à chaque étape, les « arguments de preuve » que nous fournissons sont acceptables dans le contexte courant,
- nous soumettre ce qu'il reste à prouver après que nous ayons donné une commande pour avancer dans la preuve et que cette commande ait donc été acceptée.

Faites un Run to cursor à la fin de ce que vous venez d'écrire dans le fichier. Cette manipulation sera à faire à chaque fois que nous entrerons une nouvelle commande, une nouvelle ligne. Aussi, désormais nous le répéterons plus.

Q 3.6. En regardant l'énoncé de notre théorème (et notre preuve papier), que devons-nous faire ?

Q 3.7. Quelle est la prochaine étape dans la démonstration que nous avons faite sur papier ? Sans chercher à faire preuve de beaucoup d'imagination, essayez de trouver la commande à écrire pour avancer dans la preuve. Examinez les (2) nouveaux buts générés.

Q 3.8. Quelle est la prochaine étape dans la preuve que nous avons faite sur papier ?

Q 3.9. Il ne nous reste plus à prouver, pour ce cas de la récurrence, que « *sachant que x est un naturel,* $0 = x \times 0$ ». Avec le $\times$, rappelons-le, qui représente la multiplication « mathématique » interne de Coq (il n'y a plus de référence à notre fonction `mult` ici). Une idée ?

Q 3.10. En utilisant la commande `Search`, tentez de trouver de quoi terminer la preuve du cas actuel. Cette commande est suivie (entre parenthèses) d'un motif (un squelette) de formule dans lequel on met des `_` pour les parties que l'on laisse inconnues. Par exemple :

```
Search (_ + _ = _ + _).
```

recherchera tous les théorèmes à disposition dont l'énoncé est une égalité entre deux additions.

Q 3.11. Appliquez ce théorème avec la commande `apply`.

Q 3.12. Examinez le nouveau but courant et ses hypothèses. Vérifiez qu'il correspond effectivement au second cas de notre preuve papier.

```
x, y: nat
IHy: mult x y = x * y
------------------------------------
```

```
1/1
mult x (S y) = x * S y
```

Q 3.13. L'étape suivante de notre preuve papier est un « *Par définition de mult ...* ». Quelle commande allez-vous utiliser ?

Q 3.14. Note preuve papier continue en invoquant l'hypothèse de récurrence pour conclure qu'il nous restera à prouver :

« *Par (3), (6) nous devons prouver* $x + x \times (y - 1) = x \times y$. »

La commande `rewrite` permet de réécrire le but courant en exploitant une hypothèse qui contient une égalité. Il faut donc spécifier, dans la commande, quelle hypothèse utiliser. Testez.

Q 3.15. Nous arrivons donc à une étape de preuve qui nécessite de la bête arithmétique. C'est encore une évidence que $x+x\times y = x\times(y+1)$. Quelqu'un a déjà bien dû le prouver. Cherchez...

Q 3.16. Essayez de trouver ce qui pourrait exister comme théorème à propos de la multiplication et de `S`.

Q 3.17. Essayez de trouver si le théorème statuant la commutativité de l'addition existe déjà.

Q 3.18. Utilisez le théorème de commutativité de l'addition sur le but courant. Pensez qu'il s'agit d'une égalité.

Q 3.19. Concluez le but courant.

Pensez à sauvegarder votre programme dans le fichier (menu File » Save ou touches Alt + S).

Q 3.20. Énoncez la propriété que $mult\ 2\ 3 = 2 \times 3$.

Q 3.21. Une première solution de preuve est le calcul. `mult` étant une fonction, nous pouvons l'exécuter avec la commande `compute.`

Q 3.22. À votre avis, que se passerait-t-il si nous tentions de démontrer à la place :

$$mult\ 3000000\ 2000000 = 3000000 \times 2000000$$

de cette façon ?

Q 3.23. Proposez une autre démonstration, qui tienne la route pour 3000000 et 2000000.

4 Conclusion

Nous n'avons utilisé ici qu'une **infime** partie de la puissance de l'outil Coq. En particulier, nous n'avons utilisé aucune procédure automatique de recherche de preuve. Le but était de montrer qu'il est possible de rédiger des démonstrations formellement, sans magie, en comprenant toutes les étapes élémentaires réalisées.

Dans une certaine mesure, Coq est capable de résoudre certains buts tout seul. Cela permet d'alléger un peu le labeur. Il est également possible de définir de nouvelles tactiques qui vont « essayer des trucs ». De même, il existe bien d'autres tactiques, adaptées à différentes formes de buts ou d'hypothèses.

Il n'en reste pas moins que ce ne sera pas Coq qui fera, tout seul, des démonstrations quelque

peu compliquées. Rien que notre théorème d'aujourd'hui, il ne peut pas le démontrer avec la tactique `auto`.

Nous n'avons également utilisé qu'une toute petite partie de l'expressivité de la logique sous-jacente. Il est possible de définir des objets bien plus compliqués, en utilisant des types dépendants, des prédicats inductifs et bien d'autres choses encore. Cela permet d'exprimer des propriétés plus subtiles sur des objets plus subtils.

S'il vous reste du temps ou pour continuer après la séance.

5 Pour quelques théorèmes de plus

Précédemment, nous avons vu comment transcrire de manière formelle une démonstration que nous avions faite sur papier. **Dans une certaine mesure**, Coq peut aussi nous guider lorsque l'on souhaite faire une preuve, dont on a une vague intuition, sans l'avoir préalablement faite à la main.

Nous allons illustrer ce propos en prouvant quelques théorèmes en se laissant guider par Coq. Mais attention, **en règle générale**, se lancer dans une preuve compliquée sans avoir une idée de son schéma n'aboutit pas à grand chose : comme il faut réfléchir avant de coder, il faut réfléchir avant de prouver.

Q 5.1. Énoncez le lemme `O_mult_O_n` disant que $\forall n, 0 = mult\ 0\ n$.

Q 5.2. Sur la base des commandes que vous avez déjà vues, essayez de faire la démonstration de cette propriété par induction.

Indication : lorsqu'une hypothèse correspond exactement au but à prouver, il faut invoquer la commande `assumption` qui dit à Coq « *regarde dans les hypothèses, il y en a une qui colle exactement* ».

Q 5.3. Revenons sur notre dernier lemme et sa démonstration. Nous l'avons faite par induction, comme suggéré dans l'énoncé. Mais, n'avons-nous pas démontré le théorème de correction de `mult` ? Ne peut-il pas nous servir ? Essayez.

Q 5.4. Énoncez le lemme `mult_1_n_n` disant que $\forall n, mult\ 1\ n = n$. Démontrez-le par induction.

Q 5.5. Reprenez la démonstration du théorème précédent, en exploitant le lemme de correction de `mult`.

6 Un dernier pour la route

Q 6.1. Énoncez le théorème statuant que `mult` est commutative.

Q 6.2. Commencez sa preuve jusque là où vous pouvez. Pour le sport et surtout pour pouvoir illustrer plus tard quelques aspects intéressants de la preuve de programmes, **n'utilisez pas** le lemme de correction (`mult_correct`) de `mult` que nous avons prouvé en tout début de TD.

Indication : faites une induction sur `y`.

Q 6.3. Recherchez de l'aide pour prouver le but où vous êtes (certainement) resté bloqué. Qu'en concluez-vous ?

Q 6.4. Pour ne pas rallonger le TD plus encore, nous allons faire une **énorme** entorse : nous allons définir le lemme qui nous manque comme un **axiome** ! Résultat : nous n'aurons pas besoin de le prouver. Pourquoi est-ce une **énorme** entorse ?

Q 6.5. Sachant que l'on déclare un axiome avec la même syntaxe qu'un théorème, en remplaçant le mot-clef `Lemma` par... `Axiom`, définissez l'axiome précédemment identifié. Vous devrez remonter dans votre fichier, au-dessus de notre lemme en cours de preuve pour y insérer cet axiome.

Petit défi : Dans son énoncé, inversez les opérandes de `=` par rapport à ce qui vous arrange le mieux dans votre but laissé en plan.

Q 6.6. Muni de votre nouvel allié en la personne de l'axiome, retournez terminer la preuve laissée en plan.