TME 1 : Preuves Arithmétiques

1 Prise en main

Nous utiliserons *ProofGeneral*, qui est une extension de l'éditeur de texte Emacs. Lancez `emacs tme1.v`. *ProofGeneral* doit se lancer avec `coq-mode` actif. Les raccourcis les plus courants de *ProofGeneral* sont les suivants :

- `C-c C-l` réorganise les fenêtres d'Emacs pour montrer toutes les informations.
- `C-c C-n` avance d'une commande dans la preuve courante.
- `C-c C-u` recule d'une commande dans la preuve courante.
- `C-c C-RET` avance dans la preuve jusqu'à la position du curseur.

```
Print nat.
Print bool.
Print plus. Print Nat.add.
Print mult. Print Nat.mul.
Search (_ = _ + 0).
Search (S (?X + ?Y) = ?X + S ?Y).
Compute (2 + 2).
Check (2 + 2 = 4).
```

2 Preuves arithmétiques sur l'addition

On suppose défini comme dans librairie standard de Coq les notions d'entiers, d'addition et de multiplication. Les tactiques suivantes, vues en cours, sont suffisantes pour compléter les preuves de cette partie :

- `induction x` lance une preuve par induction structurelle sur `x`. Dans le cas des entiers, cela revient à faire une preuve par récurrence.
- `intro` ajoute comme hypothèse la première variable quantifiée universellement dans le but de preuve. `intros` est une variante qui répète `intro` autant que possible.
- `simpl` applique le processus d'évaluation symbolique sur le but de preuve.
- `reflexivity` invoque la réflexivité de l'égalité (sur des buts comme `x = x`) pour terminer une preuve.
- `rewrite l` réécrit l'égalité définie dans le lemma `l` dans le but de preuve. La réécriture instancie par elle-même les arguments du lemme. Par défaut, elle remplace le membre de gauche de l'égalité par celui de droite. Il est possible d'appliquer une réécriture en sens inverse avec `rewrite <- l`.

1. Prouver que l'addition est associative sur les entiers : $(a + b) + c = a + (b + c)$.
2. Expliquer pourquoi Coq affiche $a + b + c$ plutôt que $(a + b) + c$.
3. Dans cette question, nous voulons prouver que l'addition est commutative : $a + b = b + a$. Nous commençons par établir deux résultats intermédiaires vus en cours.
   (a) Prouver que $n = n + 0$.
   Dans la suite, ce lemme est appelé `plus_n_O`.
   (b) Prouver que $(n + m) + 1 = n + (m + 1)$ (avec le successeur défini sur les entiers en Coq, `S (n + m) = n + S m`).

Dans la suite, ce lemme est appelé `plus_n_Sm`.

(c) En utilisant les deux résultats précédents, montrer que l'addition est commutative.

4. Montrer que $(a + b) + c = (a + c) + b$.

3 Preuves arithmétiques sur la multiplication

1. Définir récursivement la multiplication par récurrence sur son premier argument :
   `Fixpoint multiplication (a b : nat) : nat := ...`
2. Tester la multiplication avec `Compute`.

   Dans la suite, `*` désigne la multiplication sur les entiers telle que définie par Coq.

3. Montrer que zéro est absorbant pour la multiplication : $a * 0 = 0$.
4. Montrer la distributivité dans le cas simple suivant : `a * (S b) = a + a * b`.
   Il est conseillé d'utiliser certains des lemmes définis en Section 2.
5. En utilisant les deux résultats précédents, montrer que la multiplication est commutative : $a * b = b * a$.

   Il est souvent possible de réécrire un lemme de plusieurs façons dans les preuves qui suivent, et l'instanciation automatique de Coq peut nécessiter un guide manuel. `rewrite lemma with e1 ... en` permet de préciser à la tactique `rewrite` que `lemma` doit utiliser les arguments `e1 ... en`[1].

6. Montrer que la multiplication est distributive à droite : $a * (b + c) = a * b + a * c$.
7. Montrer que la multiplication est associative : $a * (b * c) = (a * b) * c$. L'introduction d'un lemma intermédiaire prouvant la distributivité à gauche de la multiplication est conseillé.

4 Parité

1. La négation booléenne est définie dans la librairie standard via `negb`. Prouver que `negb` est involutive, i.e., `negb (negb b) = b`.
2. Définir la fonction de parité booléenne par récurrence, en utilisant `negb`.
3. Tester cette fonction pour $n = 42$ et $n = 43$.
4. Prouver les lemmes suivants :
   (a) `even (2 * n) = true`,
   (b) `even (2 * n + 1) = false`.

1. Il est aussi possible d'utiliser `pose proof lemma e1 ... en` pour ajouter l'égalité définie par `lemma`, appliquée aux arguments `e1 ... en`, dans les hypothèses de la preuve