IN 101 - TD 07

Solution

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 Visual Studio Code qui dispose d'un mode dédié 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 devrez vous connecter, comme pour les autres TDs, à `byod.ensta.fr` en utilisant Visual Studio Code car il faut que Coq soit disponible sur la machine où vous exécuterez ce TD.

En cas d'impossibilité de se connecter à `byod.ensta.fr`, il est possible de tester Coq dans un navigateur en se rendant à l'adresse `https://coq.vercel.app/scratchpad.html` (ne semble pas fonctionner sous Mac/Safari mais fonctionne avec Firefox).

Q 2.1. Lancez Visual Studio Code et créez un nouveau fichier nommé `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 donnée en cours qui suit et assurez-vous de bien la comprendre :

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

- 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 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.
```

Solution

La première ligne est en quelque sorte de l'administratif au même titre que le `#include` de C dont vous avez désormais l'habitude.

Une fonction récursive commence par le mot-clef `Fixpoint` suivi de ses paramètres. Le type des paramètres peut-être donné avec des `:` et `nat` représente les entiers naturels (comme le `unsigned int` de C).

Même s'il y a la possibilité d'utiliser la syntaxe habituelle pour les entiers, nous allons privilégier leur représentation interne en Coq. Un *entier naturel* est soit `O` (pour 0), soit « le successseur » `S` d'un *entier naturel* n : `S n`. Ainsi 3 est représenté par `S (S (S O))`.

Cette représentation nous permet de ne pas parler de la soustraction sur les entiers naturels. Comment la définir formellement, lorsque dans $x - y$ nous avons $x < y$? Dans les naturels, il n'y a pas de négatifs ! Donc autant évincer la question en évitant de se poser le problème.

Cette représentation va également nous permettre de travailler très simplement par cas, donc par récurrence sur les naturels. Et justement, l'analyse par cas, ressemblant au `switch` de C est réalisée par la construction `match`.

- `match y with` signifie « *regardons quelle est la forme de* `y` » *et effectuons un traitement pour chacun des cas possibles* ». Viennent ensuite les cas possibles. À gauche de la `=>` se trouve la forme d'un cas, avec le résultat à retourner à droite. Chaque cas est séparé par une barre verticale.
- `| O => O` dit « *si* `y` *est* `O` *alors retourner le résultat* `O` ».
- `| S n => x + (mult x n)` dit « *si* `y` *est de la forme* `S` *de quelque chose que j'appelle* `n`*, retourner la valeur de* `x` *plus le résultat de la fonction* `mult` *appliquée à* `x` *et* `n` ».

À une syntaxe nouvelle près, on retrouve bien l'algorithme énoncé en pseudo-code en cours et rappelé en début de sujet. Remarquons que nous n'avons pas d'instruction `return` : elle est implicite.

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 Visual Studio Code, faites un clic droit à la fin de la fonction pour faire apparaître le menu contextuel et sélectionnez Interpret to point (un raccourci clavier existe et est mentionné dans l'entrée de menu). Si vous travaillez avec Coq dans un navigateur, utilisez ⌥+↑ et ⌥+↓ pour vous déplacer à une ligne en la faisant interpréter par Coq.

Solution

Rien à signaler, pas de message d'erreur, Coq est content et la fonction est définie. Nous pouvons d'ailleurs le vérifier en tapant :

```
Check mult.
```

puis en rappelant le menu Interpret to point à la fin de la ligne que nous venons de taper. Coq affiche en retour :

```
mult
 : nat -> nat -> nat
```

qui indique que `mult` est connue et est une fonction qui prend deux naturels et retourne un naturel (dans une syntaxe possiblement inhabituelle pour vous si vous n'avez pas fait d'OCaml en prépa).

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.
```

Solution

La bibliothèque standard de Coq contient la définition de la multiplication et d'un certain nombre de ses propriétés mathématiques. Cette formalisation est un immense travail sur lequel les utilisateurs peuvent s'appuyer. Ainsi, `*` représente cette multiplication.

Notre énoncé de théorème dit quoi ? Il dit que « *pour tout entier* `x` *et pour tout entier* `y`*, le résultat de notre* **fonction** `mult` *(qui est bien un* **programme***) doit être égal à ce que la multiplication "des maths" donne pour* $x \times y$ ».

Dit autrement, nous allons prouver que notre **programme** est conforme à la spécification de la multiplication mathématique. Soyons bien conscient que notre programme, même s'il fait une multiplication, il **n'utilise pas** la multiplication. Et c'est bien pour cela que l'on voudrait s'assurer qu'il fait bien ce que l'on attend ... et prétend !

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 Interpret to point 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.

Solution

Coq nous affiche le retour suivant :

```
------------------------------------
1/1
forall x y : nat, mult x y = x * y
```

pour indiquer que nous sommes en mode « preuve » et qu'il attend que nous lui disions quoi faire. Il nous montre le **but courant** sous la barre horizontale. Au-dessus de cette barre se trouvent les **hypothèses** actuellement connues ou déduites. Là, nous voyons clairement qu'il n'y en a pas. Normal, nous n'avons encore rien fait, donc nous avons juste un but !

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

Solution

Cet énoncé nous donne 2 premières hypothèses : « *Soit x un entier naturel, soit y un entier naturel* ». Il faut donc poser réellement ces hypothèses en « cassant notre formule ».

Le « soit ...» des maths se dit `intro` en Coq. Nous allons donc faire monter nos `x` et `y` dans les hypothèses : on les `intro`-duit :

```
intro x.
intro y.
```

Le nom qui suit `intro` sert juste à nommer l'hypothèse et est optionnel. Par défaut, lorsque ce qui est « `intro` » est une variable, Coq nomme l'hypothèse avec le nom de la variable. Donc, ici, donner le nom est tout à fait inutile, Coq aurait naturellement attribué le même nom.

Cliquons Interpret to point à la fin de notre première nouvelle ligne et Coq nous affiche :

```
x : nat
----------------------------------
1/1
forall y : nat, mult x y = x * y
```

Cela nous montre qu'il a accepté notre ordre d'`intro` et qu'il a fait le travail de mettre `x` parmi les hypothèses (comme demandé), nous laissant maintenant le but à prouver `forall y : nat, ....`

Avançons jusqu'à la seconde ligne que nous avons rajoutée et sans surprise Coq en fait de même avec `y` pour nous afficher en retour :

```
x, y: nat
----------------------------------
1/1
mult x y = x * y
```

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.

Solution

La prochaine étape est de faire une induction sur y. « Sans chercher à faire preuve de beaucoup d'imagination », la commande est simplement `induction y.`

Coq nous génère alors 2 buts, dont l'un devient le but courant et l'autre est en mis suspens et nous sera soumis quand nous aurons prouvé le premier. Ces buts correspondent à la propriété que nous devons prouver instanciée par les 2 cas possibles de formes d'entiers naturels :

- `mult x 0 = x * 0`,
- `mult x (S y) = x * S y` (qui peut se lire comme *mult* $x\ (y+1) = x \times (y+1)$.

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

Solution

C'est « *Par définition de mult ...* ». Cela revient en fait à dire que le code de la fonction `mult` que nous avons écrite dit explicitement que « *si y vaut 0 alors le résultat est 0* ». Pour dire à Coq de **simplifier** le but courant en utilisant la définition (le code) des fonctions qu'il connaît, il faut invoquer la tactique `simpl`.

On Interpret to point à la fin de cette nouvelle ligne et Coq nous affiche l'état de la preuve :

```
x: nat
----------------------------------
1/1
0 = x * 0

2/2
mult x (S y) = x * S y
```

Il a remplacé `mult x 0` par ce que retourne (le code de) la fonction `mult` dans le cas où `y` vaut 0 : à savoir 0.

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 ?

Solution

C'est une évidence, nous le savons toutes et tous. Pour autant, dans nos hypothèses nous n'avons rien qui nous permette d'asséner cette « évidence » !

Puisque Coq dispose d'une bibliothèque standard, il doit bien y avoir quelqu'un qui a démontré cette évidence. Nous devons bien arriver à trouver un tel théorème et nous en servir pour achever notre but.

Nous agissons exactement de la même façon que lorsque nous programmons : au lieu de dire « *est-ce qu'il n'existerait pas une fonction qui fait le travail ?* », nous disons « *est-ce qu'il n'existerait pas un théorème qui fait le travail ?* ».

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.

Solution

Nous allons donc rechercher s'il existe un théorème de la forme « 0 = truc × 0 » : `Search (0 = _ * 0).` Ce à quoi Coq répond par une longue liste dont la première entrée nous semble un excellent candidat.

```
mult_n_O: forall n : nat, 0 = n * 0
ZL0: 2 = 1 + 1
Nat.add_comm: forall n m : nat, n + m = m + n
...
```

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

Solution

Nous rajoutons donc la ligne contenant `apply mult_n_O.` et Coq ayant constaté qu'elle clôt le but nous fait automatiquement basculer sur le but qui était en suspens.

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
```

Solution

Le second cas de notre preuve papier était :

« *Cas $y > 0$, prouvons que $mult(x, 0) = x \times 0$* »

Coq nous a bien **automatiquement** généré l'obligation que nous avons désormais à prouver puisque nous sommes dans une preuve par induction. Puisque nous sommes dans le cas $y > 0$, nous voyons que Coq a remplacé notre y « papier » par `S y`. Autrement dit, il rend explicite que notre second argument est un « successeur de quelque chose qu'il nomme `y` ». Ainsi, on n'aura pas à parler de $y - 1$, ce sera simplement y.

On retrouve dans les hypothèses... l'hypothèse d'induction (ou de récurrence) :

```
IHy: mult x y = x * y
```

qui est nommée `IHy`. C'est justement celle-là que notre preuve papier va utiliser d'ici peu.

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

Solution

Comme précédemment, nous appliquons la commande `simpl` et Coq déroule ce qu'il sait du code de `mult`. Nous obtenons donc l'état suivant :

```
x, y: nat
IHy: mult x y = x * y
------------------------------------
1/1
x + mult x y = x * S y
```

Oh surprise ! Il a effectivement remplacé `mult x (S y)` par `x + mult x y`. Exactement ce que nous avons écrit dans notre preuve papier, à la différence que notre $y - 1$ papier est simplement `y` dans notre preuve Coq. En effet, nous l'avons vu à la question précédente, en raisonnant sur la structure interne des entiers naturels, Coq nous permet d'éviter de parler de soustraction et lorsque nous avons $y > 0$ alors cela signifie juste que y est de la forme `S` *qqchose*.

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.

Solution

Par défaut, lorsqu'une hypothèse `H` est de la forme $a = b$, la commande `rewrite H` réécrit dans le but courant les occurrences de a par b. Ça tombe bien, c'est exactement dans ce sens que nous voulons utiliser l'hypothèse de récurrence. Nous faisons donc `rewrite IHy.`

```
x, y: nat
IHy: mult x y = x * y
------------------------------------
1/1
x + x * y = x * S y
```

et notre but devient exactement ce que nous avions prédit dans notre preuve papier (avec toujours notre y papier qui est `S y` dans notre preuve Coq).

Si dans l'hypothèse l'égalité était dans l'autre sens, il aurait fallu spécifier à `rewrite` de faire la réécriture de la droite vers la gauche. On aurait alors juste dû expliciter le sens de la réécriture par `rewrite <- IHy.` car par défaut le sens est `->`.

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...

Solution

Nous allons faire une recherche avec la commande `Search`. Une solution simple est d'invoquer :

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

en mettant juste un `_` à droite, sans même chercher à préciser `= _ * _`, comptant sur le fait qu'il n'y aura sans doute pas des milliers de résultats à lire. Effectivement, on obtient ... rien ! Non, personne n'a déjà prouvé un truc de cette forme dans la bibliothèque standard. Il va donc falloir réfléchir un peu.

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

Solution

La commande `Search (_ = _ * S _).` nous donne quelque chose d'intéressant :

```
mult_n_Sm: forall n m : nat, n * m + n = n * S m
```

c'est presque ce que l'on doit prouver sauf qu'à gauche, la multiplication est en premier alors que dans notre but elle est en second (le théorème dit `n * m + n = ...` or nous avons a besoin de `n + n * m = ...`).

Mais, nous savons bien que l'addition est commutative, donc il doit y avoir un théorème qui nous permet de réécrire notre but courant pour permuter les opérandes de notre addition.

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

Solution

Cherchons avec `Search (_ + _ = _ + _).` :

```
mult_n_Sm: forall n m : nat, n * m + n = n * S m
mult_n_Sm: forall n m : nat, n * m + n = n * S m
Nat.add_comm: forall n m : nat, n + m = m + n
Plus.plus_Snm_nSm_stt: forall n m : nat, S n + m = n + S m
...
```

et nous trouvons effectivement `Nat.add_comm` qui fait exactement le travail.

Nous aurions pu affiner la recherche en remplaçant des `_` par des `?a`, `?b`. Si un tel « joker » apparaît plusieurs fois dans la formule recherchée, alors dans les résultats il doit apparaître avec la même valeur à tous les emplacements. Par exemple :

```
Search (?a + ?b = ?b + ?a).
```

nous donne uniquement en résultat de recherche le théorème dont nous avons besoin. La recherche est vraiment ciblée pour le coup !

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

Solution

Puisque c'est une égalité, nous pouvons utiliser `rewrite` pour réécrire le but courant en faisant `rewrite Nat.add_comm.` :

```
x, y: nat
IHy: mult x y = x * y
----------------------------------
1/1
x * y + x = x * S y
```

Le but a désormais exactement la forme du théorème `mult_n_Sm` que nous avons trouvé précédemment.

Q 3.19. Concluez le but courant.

Solution

Il ne nous reste plus qu'à appliquer le théorème `mult_n_Sm` par la commande `apply mult_n_Sm.` Coq nous informe alors que *Proof finished.* Afin de clôturer définitivement la preuve et de sauvegarde notre théorème pour pouvoir s'en servir dans d'autres preuves, il faut finir par la commande `Qed.` En résumé, nous avons la preuve suivante :

```
Lemma mult_correct : forall x y : nat, (mult x y) = (x * y).
Proof.
  intro x.
  intro y.
  induction y.
  - (* Cas y = 0 *)
    simpl.
    apply mult_n_O.
  - (* Cas y > 0 *)
    simpl. rewrite IHy.
    rewrite Nat.add_comm.
    apply mult_n_Sm.
Qed.
```

Remarquez l'utilisation des ≪ bullets ≫ `-` pour structurer les sous-cas de preuve et des commentaires `(* pour documenter le code *)`.

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

Solution

Il suffit de déclarer un nouveau lemme et d'entrer en mode preuve :

```
Lemma trois_fois_deux : (mult 3 2) = 3 * 2.
Proof.
```

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

Solution

Nous invoquons donc la commande `compute` qui nous laisse le but suivant à prouver :

```
----------------------------------
1/1
6 = 6
```

La preuve de cette formule tient à la réflexivité de l'égalité. La tactique adéquate en Coq s'appelle, sans originalité, `reflexivity`. Nous pouvons ainsi clore la démonstration par le `Qed` final.

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 ?

Solution

Coq va entrer dans l'exécution de la fonction `mult`, avec 2 millions d'appels récursifs, ce qui va prendre un temps déraisonnable... et de toute façon terminer pas un débordement de pile, donc un échec.

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

Solution

Puisque nous avons démontré au cours de ce TD que $\forall x, y \in \mathbb{N}$, $mult(x, y) = x \times y$, c'est devenu un théorème **au même titre** que ceux précédemment existants. Nous n'avons qu'à nous en servir pour démontrer notre proposition.

En effet, notre proposition est exactement le corps de notre théorème, avec x remplacé par 3000000 et y par 2000000. Et là, la démonstration est immédiate, sans longs et coûteux calculs.

```
Lemma beaucoup_fois_beaucoup : (mult 3000000 2000000) = 3000000 * 2000000.
Proof.
  apply mult_correct.
Qed.
```

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 disant que $\forall n, 0 = mult\ 0\ n$.

Solution

Il nous faut utiliser la construction `Lemma`, donner un nom à notre futur théorème et écrire la formule Coq correspondante.

```
Lemma O_mult_O_n: forall n, 0 = mult 0 n.
  Proof.
```

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

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* ≫.

Solution

Cette preuve va se faire par induction sur `n`. Comme précédemment, nous montons `n` dans les hypothèses avec un `intro`. Puis nous invoquons l'`induction` (sur) `n`.

À la fin du second cas, le but à prouver est exactement l'hypothèse `IHn` donc il suffit d'invoquer `assumption`. Si nous voulions conclure en désignant manuellement l'hypothèse à utiliser, nous pourrions le faire en disant `exact IHn.`, mais nous n'allons pas nous plaindre de bénéficier d'un peu d'automatisation.

```
Lemma O_mult_O_n: forall n, 0 = mult 0 n.
  Proof.
    intro n. induction n.
    - simpl. reflexivity.
    - simpl. assumption.
  Qed.
```

Q 5.3. Énoncez et démontrez le lemme disant que $\forall n, mult\ 1\ n = n$.

Solution

Cette preuve se fait également par induction sur `n`. Elle nécessite de réécrire un des buts courants par l'hypothèse d'induction `IHn` pour conclure par la `reflexivity`.

```
Lemma mult_1_n_n : forall n : nat, mult 1 n = n.
Proof.
```

```
  intro. induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.
```