TP: Introduction à Coq

Olivier Hermant

septembre 2020

1 Introduction

1.1 Présentation

Coq est un assistant à la preuve qui s'appuie sur le Calcul des Construction Inductives. Le calcul des constructions est un système de types très puissant pour les λ-termes, avec la garantie que tous les λ-termes typables normalisent fortement (*i.e.* toute suite de β-réductions au départ d'un terme est finie). A ce λ-calcul typé ont été rajoutés des types dits *inductifs* qui permettent une grande facilité d'utilisation.[1]

Il est possible d'exprimer, de façon quasiment native, les entiers (unaires, mais aussi binaires), les listes, les vecteurs, ... Certains considèrent et utilisent même Coq comme un langage de programmation à part entière: on peut définir pratiquement toutes les fonctions,[2] le souci principal est l'efficacité du code (il s'exécute vraiment très lentement par rapport à un programme en C ou en OCaml par exemple). L'avantage majeur réside bien évidemment dans la richesse du système de type et la possibilité de spécifier très précisément (par exemple à l'aide d'un type très raffiné) d'énoncer et de prouver des propriétés sur la fonction que l'on écrit.

Nous allons nous intéresser dans un premier temps à la logique propositionnelle, puis à la logique des prédicats. Notez que, dans le calcul des constructions, les types sont eux-même des termes, et on peut donc aussi les abstraire et les passer en argument.[3]

Par exemple, nous pouvons définir une fonction identité polymorphe:

$$\lambda\alpha : \texttt{Type}.\lambda x : \alpha.x$$

ou encore le type de vecteurs dépendant suivant

$$\lambda n : \texttt{Type}.\texttt{array n}$$

où `array n` est le type des vecteurs de longueur `n`. Il est aussi possible de définir un type de vecteurs dépendant et polymorphe, etc, etc.

Notons qu'une forme limitée de polymorphisme est possible en Haskell, le polymorphisme "à la ML". Dans ce cas limité il est possible de laisser implicite la quantification universelle sur le type.

1.2 Démarrer Coq

L'IDE `coqide` peut être lancée en ligne de commande, ou par le menu. `coqtop` est un mode interactif en ligne de commande.

Si vous préferez utiliser `emacs` (ce que je peux comprendre), le mode majeur `proofgeneral` est installé, mais n'est que peu décrit ci-dessous. Il peut se lancer en ligne de commande avec `emacs`. Tapez votre code, et sauvegardez le avec l'extension "`.v`" (par exemple `first_order_logic.v`). Des options supplémentaires apparaîtront alors.

Les étapes pour écrire un fichier Coq correct sont les suivantes:

1. définir les symboles utilisés.

[1] ainsi qu'une plus grande puissance théorique. Par exemple, dans le calcul des constructions, il n'est pas possible de prouver que 1 est différent de `0`, ce qui devient possible dans le calcul des constructions inductives (véridique).

[2] même si le langage n'est pas Turing-complet, seules des fonctions très vicieuses ne rentrent pas dans le moule.

[3] le plus étonnant dans tout cela n'est pas la *possibilité* de le faire, mais que le système continue à rester cohérent, sans contradiction. Nous n'irons pas jusque là.

2. définir les formules à prouver.

3. écrire la preuve.

4. demander au noyau de Coq de vérifier que cette preuve est correcte. En d'autre termes, c'est le moment où l'on demande de vérifier que la preuve (qui est un λ-term, même s'il n'apparaît pas explicitement sous cette forme) est bien typée.

1.3 Utiliser Coq

1.3.1 Les commandes, leur interprétation

Toutes les commandes Coq se terminent par un point ".". Voici une première commande: `Check`, elle sert à vérifier le type des constantes et des définitions. Par exemple, `Check 3.` (avec la majuscule et le point) est une commande valide.

Il faut ensuite lancer l'interpréteur afin d'évaluer les commandes que l'on a écrites.

- dans `coqide`:
  - écrire la commande dans la fenêtre de gauche.
  - Aller dans le menu `Navigation`, `Forward`.
  - l'instruction se retrouve surlignée en vert, indiquant qu'elle a été exécutée.
  - la fenêtre en bas à droite montre alors la réponse de Coq, en l'occurrence `3 : nat` (et un message d'avertissement), ce qui veut dire que `3` a le type `nat`. Ce message d'avertissement peut être évité en faisant cette requête de manière interactive : sélectionner the charactère `3`, aller dans `Queries -> Check`.
- dans `emacs`, aller voir le menu `Proof-General -> Next Step`. Cela lance l'interpréteur et exécute la première instruction. Dans le cas de `Check 3.` l'effet devrait être:
  - une division de la fenêtre d'`emacs`.
  - le surlignage en bleu de `Check 3.`
  - dans la sous-fenêtre inférieure, la réponse de Coq, qui devrait être `3 : nat`.

Pour s'amuser un peu, tenter l'instruction `Check nat.`: eh oui, le type aussi fait partie des λ-termes du calcul des constructions inductives ! Vous pouvez aussi vérifier le type du type de `nat`, le type du type de son type, etc, etc. Il n'y a pas de limite, dans le *calcul des constructions inductives avec univers*.[4]

1.3.2 Former des propositions

Une proposition se forme à partir:

- de symboles de propositions `A`, `B`, `C`, ... Si on veut les utiliser, il faut les définir, à l'aide de la commande `Variable nom: type`. En l'occurence, pour définir un symbole de *proposition* `A`, on écrira `Variable A: Prop` (le type des propositions s'appelle `Prop`).
- de connecteurs logiques, dont voici la manière de les écrire en Coq:

$\wedge$	$\vee$	$\Rightarrow$	$\neg$
`/\`	`\/`	`->`	`~`

La propositions $A \Rightarrow B \Rightarrow A$ s'écrira donc: `A -> (B -> A)` ou, plus simplement, `A -> B -> A`.

[4] Remarquez que la règle de typage `Type: Type` est un brin incestueuse. Elle l'est réellement. (Girard, 1972) puis (Miquel, 2000) ont démontré que l'on pouvait encoder tout un tas de paradoxes, par exemple celui de l'ensemble de tous les ensembles qui ne se contiennent pas, ou celui du fou qui repeint son plafond. En fait, dans la réalité, Coq cache sous le tapis une hiérarchie d'univers. On a, formellement, $\texttt{Type}_i : \texttt{Type}_{i+1}$, mais l'indice i n'est pas affiché, a moins d'utiliser la commande `Set Printing Universes.` ou d'aller chercher le menu `View` de `CoqIde`.

1.3.3 Enoncer un théorème ou un lemme

Cela se fait avec le mot clef `Theorem` (resp. `Lemma`):

```
Theorem nom_du_theoreme : proposition_a_prouver.
```

Par exemple: `Theorem identite : A -> A.` Auparavant il faut avoir défini `A` elle-même, soit comme une constante globale (`Variable A: Prop.`), mais nous allons éviter de travailler ainsi et généraliser un peu.

Ainsi, nous allons systématiquement quantifier de manière locale sur tous les objets du théorème: dessus :

```
Theorem identite : forall A: Prop, A -> A.
```

Cette quantification est possible en Coq, dont la puissance logique est d'ordre supérieur[5].

Les lemmes s'énoncent de la même manière (mot-clef `Lemma`).

1.3.4 Prouver un théorème ou un lemme

Une fois le théorème énoncé, il faut le prouver. Une fois que Coq l'a vérifié, vous remarquerez qu'apparaît ceci:

```
1 subgoal


============================
forall A: Prop, A -> A
```

Il s'agit de la proposition que vous devez prouver. La barre `=======` représente tout simplement le "turnstyle" $\vdash$ du séquent. Vous devez donc prouver (en déduction naturelle) le séquent: $\vdash \forall A, A \Rightarrow A$.

Nous ne savons pas (encore) faire ceci, à cause de la quantification universelle en tête, mais il est très simple de s'en débarrasser – il suffit de demander à Coq d'effectuer `intro.` et nous nous retrouvons à démontrer ceci :

```
1 subgoal
A: Prop
============================
A -> A
```

Il s'agit, sous l'hypothèse explicite que A est une proposition que nous omettrons par la suite[6], de la notation que Coq a pour le séquent

$$\vdash A \Rightarrow A$$

Pour démontrer ce séquent, comme vous le savez, il suffit d'appliquer la règle $\Rightarrow$-intro puis un axiome. En Coq, $\Rightarrow$-intro se dit `intro.` Essayez.

Vous devez maintenant prouver le séquent $H : A \vdash A$ (Coq a automatiquement nommé `H` l'hypothèse que vous avez introduite à gauche du séquent). En Coq:

```
1 subgoal
  A : Prop
  H : A
  ============================
   A
```

Il suffit maintenant de dire à Coq d'utiliser l'hypothèse `H`, par l'instruction `apply H.` ou bien directement d'appliquer la règle axiome avec la commande `assumption.`. Une fois que Coq vous dit `proof completed`, il ne vous reste plus qu'à lui dire "CQFD" (`Qed.`, en latin). Vous pouvez aussi utiliser la commande `Save.` (Il n'y a pas de différence à ce niveau). Cela a pour effet de demander à Coq de vérifier la preuve, c'est à dire, en particulier de produire le λ-terme associé. Vous pouvez voir le λ-terme explicite en utilisant la commande `Print nom_du_theoreme.` ou par l'option adéquate du menu `Queries` de `coqide`.

Lorsque vous devez prouver plusieurs sous-arbres (par exemple après un $\wedge$-intro, il y a deux prémisses à démontrer), Coq vous demande de les prouver les uns après les autres (lorsqu'un sous-arbre est prouvé, il bascule sur le sous-arbre de preuve suivant).

Il est possible de naviguer entre les sous-arbres avec la commande `Focus`.

[5] comprendre : qui permet la quantification sur des objets de type "élevé".

[6] elle sera donc implicite pour nous, et cela n'aura aucun impact sur le travail de cette session, qui ne s'attaque qu'au premier ordre

1.3.5 Généraliser un peu

Notre théorème `identite` est, pour rappel

```
Theorem identite : forall A: Prop, A -> A.
```

Il est donc *générique* en ce sens qu'il peut s'appliquer à *n'importe quelle* proposition A, ce qui peut servir dans d'autres démonstrations, par exemple :

```
Goal forall B : Prop, (B -> B) -> (B -> B).
intro.
apply (identite (B -> B)).
```

On a ainsi instancié le A du théorème `identite` par $B \Rightarrow B$ – noter au passage les parenthèses pour appliquer le théroème `identite` (vu comme une fonction !) à l'argument $B \Rightarrow B$. Le mot-clef `apply` demande simplement d'appliquer le *résultat* de l'utilisation du théorème.

2 Preuves en logique propositionnelle

2.1 Les règles

Voici la table de correspondance entre les règles de la déduction naturelle et les instructions de Coq:

	$\wedge$	$\vee$	$\Rightarrow$	$\neg$
Introduction:	`split`	`left` / `right`	`intro`	`intro`
Elimination:	`elim`	`destruct`	`apply`	`elim`

Les commandes d'application des règles d'élimination s'utilisent avec le nom de l'hypothèse (donc: `elim H.` ou `apply H.` par exemple, car il faut dire quelle est la formule dont vous voulez éliminer le connecteur principal). Notez aussi que lorsque l'on veut, par exemple, prouver le séquent $A \wedge B \vdash C$, et que l'on applique la règle `elim` sur la conjonction $\wedge$, on construit l'arbre de preuve suivant:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash B}\wedge\text{-elim2} \qquad \dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash A}\wedge\text{-elim1} \qquad \dfrac{\vdots}{A \wedge B \vdash A \Rightarrow (B \Rightarrow C)}}{A \wedge B \vdash B \Rightarrow C}\Rightarrow\text{-elim}}{A \wedge B \vdash C}\Rightarrow\text{-elim}$$

Ainsi, on passe de devoir prouver $A \wedge B \vdash C$ à devoir prouver $A \wedge B \vdash A \Rightarrow (B \Rightarrow C)$. Après avoir fait les deux `intro` nécéssaires, on se retrouve à devoir prouver le séquent: $A \wedge B, A, B \vdash C$. La correspondance avec les règles $\wedge_e$ de la déduction naturelle n'est pas flagrante. Notez que l'on peut faire en un seul coup `elim` et les deux `intro` avec une seule commande : `destruct`.

De même, lorsque l'on utilise la commande `destruct` sur (le nom d'une hypothèse qui est) une disjonction, l'arbre de preuve construit est le suivant:

$$\dfrac{\dfrac{\vdots}{A \vee B, A \vdash C} \qquad \dfrac{\vdots}{A \vee B, B \vdash C} \qquad \dfrac{}{A \vee B \vdash A \vee B}}{A \vee B \vdash C}$$

Nous avons donc deux sous-arbres, c'est à dire deux sous-buts Coq, à prouver. C'est directement la règle $\vee$-elim. On aurait pu, avec un peu plus de difficulté, utiliser la règle `elim`.

Enfin, la commande `intros` effectue autant de fois la règle `intro` que possible (sauf pour la négation). Il est aussi possible d'introduire artificiellement certaines propositions (des lemmes, que l'on doit démontrer par la suite, *no free lunch*). Il est théoriquement possible de s'en passer, mais cela est parfois utile:

```
assert F.
```

`F` peut être n'importe quelle formule. Deux choses se passent alors: `F` est mise dans le contexte en tant qu'hypothèse supplémentaire, et l'arbre de preuve récupère une nouvelle branche, dans laquelle on doit démontrer `F`.

2.2 Propositions à démontrer

Démontrer les propositions suivantes:

1. $A \Rightarrow B \Rightarrow A$
2. $(A \Rightarrow B) \Rightarrow (C \Rightarrow A) \Rightarrow C \Rightarrow B$
3. $A \Rightarrow (A \vee B)$
4. $A \Rightarrow B \Rightarrow (A \wedge B)$
5. $(A \Rightarrow B) \Rightarrow (C \Rightarrow B) \Rightarrow ((A \vee C) \Rightarrow B)$
6. $(A \vee (B \vee C)) \Rightarrow ((A \vee B) \vee C)$
7. $\neg\neg\neg A \Rightarrow \neg A$
8. $(\neg A \wedge \neg B) \Rightarrow (A \vee B) \Rightarrow \bot$

Indications: $\bot$ se nomme `False` en Coq. Parfois, il est plus rapide d'arriver avec une preuve sur le papier, et d'essayer de la faire comprendre à Coq. D'autres fois, il peut être intéressant de laisser à Coq le soin de vous guider.

2.3 Tactiques

Coq offre de nombreuses *tactiques*, qui sont des scripts (plus ou moins compliqués) permettant de construire automatiquement un (terme de) preuve lorsque le but à prouver a une certaine forme (qui dépend de la tactique). Coq offre même un langage de tactiques `LTac` vous permettant d'écrire vos propres tactiques.

En particulier, Coq a une tactique permettant de démontrer automatiquement tous les buts de logique propositionnelle pure. Elle s'appelle `tauto` (pour *tautologie*). Note : cette tactique n'a rien à voir avec les tautologies booléennes, elle construit réellement une preuve syntaxique, sans chercher à savoir si le but est vrai ou faux.

Utilisez `tauto` (ou d'autres tactiques) pour démontrer les formules suivantes:

1. $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$
2. $(A \wedge (B \vee C)) \Rightarrow ((A \wedge B) \vee (A \wedge C))$
3. $(A \vee (B \wedge C)) \Rightarrow ((A \vee B) \wedge (A \vee C))$
4. $(A \Rightarrow B) \Rightarrow (A \wedge C) \Rightarrow (B \wedge C)$
5. $\neg(A \vee B) \Rightarrow (\neg A \wedge \neg B)$

3 Logique des prédicats

3.1 Les règles

La syntaxe pour les quantificateurs est |forall— et |exists—. Les règles correspondantes sont:

	$\forall$	$\exists$
Introduction	`intro`	`exists v`
Elimination	`apply`	`elim`

En général, lorsque l'on utilise la règle $\forall$-elim (et la règle $\exists$-intro), il faut donner le terme qui instanciera la variable quantifiée, car Coq ne peut pas la deviner dans le cas général. Soit donc un terme v, et admettons que nous avons une hypothèse `H : forall x : T, A` (pour un certain type `T`). On instancie x par v tout simplement par l'application `H v`. `H` n'est en effet rien d'autre qu'une fonction. Il suffit ensuite d'appliquer la règle:

```
apply (H v).
```

Les parenthèses sont ici importantes. Quelques fois, il peut y avoir besoin d'enchaîner les instanciations et les règles d'élimination, auquel cas il suffira d'écrire: `elim (H v)` ou `elim (H v w)` (si l'on veut instancier deux fois).

De plus, il faut faire attention à ce que toutes les variables ou les constantes introduites dans le terme v soient **bien déclarées**, soit dans le contexte courant (à gauche du $\vdash$), soit globalement. Coq impose d'utiliser uniquement des éléments dont on a *déjà prouvé/supposé l'existence*, ce qui est l'unique manière formelle de faire. Comme nous allons quantifier sur des variables que nous supposerons avec le type `nat`, il y aura un certain nombre de candidats naturels pour ce faire.

Supposons devoir prouver la formule

$$\forall x P(x) \Rightarrow P(0)$$

Il faut définir les *prédicats* et donner un type à x. C'est à dire que si on veut prouver cette formule, il faut explicitement dire que `P` est un prédicat qui prend un entier, par exemple (ce sera notre choix ici), et que `x` est un entier. Le type de `P` sera donc `nat -> Prop` (ou bien `Set -> Prop`, ou encore `Type -> Prop`, etc). Le théorème s'énoncera donc:

```
Theorem abc: forall P: nat -> Prop, (forall x: nat, (P x)) -> (P 0).
```

Lorsque l'on veut définir un prédicat à deux variables, celui-ci devra donc être du type `nat -> nat -> Prop` ou bien, ce qui est légèrement différent, du type `nat*nat -> Prop` (où `*` dénote le produit cartésien).

La preuve, elle, se fait de manière habituelle. Notez que Coq se charge tout seul de renommer les variables au cas où un conflit de nommage apparaît, notamment pour les règles $\forall$-intro et $\exists$-elim.

3.2 Formules à démontrer

Prouver les formules suivantes, vous pouvez vous servir librement des tactiques de Coq, notamment `tauto` et `auto`:

1. $[\forall x \forall y R(x, y)] \Rightarrow \forall x \forall y R(y, x)$;
2. $(\exists y \forall x P(x, y)) \Rightarrow (\forall x \exists y P(x, y))$
3. $((\forall x P(x)) \vee Q) \Rightarrow \forall x (P(x) \vee Q)$ (attention: P est un prédicat unaire (à une variable) et Q est un prédicat 0-aire, *i.e.* sans variable) ;
4. $[\exists x (\neg P(x))] \Rightarrow \neg(\forall x P(x))$.

4 Tiers-exclu – plus difficile

Coq est constructif, et à ce titre il ne peut pas démontrer $A \vee \neg A$ pour tout A. Aussi, si l'on veut utiliser ce principe controversé, il faut l'ajouter en tant qu'axiome:

```
Axiom excluded_middle : forall A:Prop, A \/ ~A.
```

On utilise un axiome comme un lemme, sauf que l'on a pas besoin de le démontrer, c'est le propre des axiomes. Par exemple, pour démontrer

```
Theorem tiers_2: forall B:Prop, (B -> B) \/ ~ (B -> B).
```

Il suffit de faire `intros. apply tiers_exclu.` Coq instanciera directement la propositions A de l'axiome par $B \Rightarrow B$. On aurait pu être plus précis et donner l'instance du tiers-exclu que l'on veut appliquer: `apply (tiers_exclu (B0 -> B0)).` de manière à indiquer que l'on a un axiome avec le tiers-exclu appliqué à $B_0 \Rightarrow B_0$. Bien sûr, le théorème `tiers_2` ci-dessus peut tout à fait être prouvé sans l'aide du tiers-exclu (exercice).

De manière générale, le tiers-exclu permet de "stocker provisoirement" des formules dans les hypothèses, alors que l'on travaille sur d'autre formules à droite du séquent. Par exemple, si l'on veut démontrer le séquent $\Gamma \vdash B$, on peut commencer par faire ceci:

$$\dfrac{\text{Ax.}\ \dfrac{}{\Gamma, B \vdash B} \qquad \dfrac{\vdots}{\Gamma, \neg B \vdash B} \qquad \dfrac{}{\Gamma \vdash B \vee \neg B}\ \text{excluded-middle}}{\Gamma \vdash B}\ \vee\text{-elim}$$

Il nous reste maintenant à prouver le séquent $\Gamma, \neg B \vdash B$: nous sommes revenus à devoir prouver B, *mais* nous avons *plus* d'informations en hypothèses, ce qui peut s'avérer utile. Un schéma que l'on retrouve souvent dans ces preuves est le suivant:

$$\dfrac{\text{Ax.}\ \dfrac{}{\Gamma, B \vdash B} \qquad \dfrac{\dfrac{\dfrac{\text{Ax.}\ \dfrac{}{\Delta, \neg B \vdash \neg B} \qquad \dfrac{\vdots}{\Delta, \neg B \vdash B}}{\Delta, \neg B \vdash C}\ \neg\text{-elim}}{\Gamma, \neg B \vdash B}\ \text{Ax.}}{\Gamma \vdash B} \qquad \dfrac{}{\Gamma \vdash B \vee \neg B}\ \text{excluded-middle}}{\Gamma \vdash B}\ \vee\text{-elim}$$

On doit maintenant démontrer le séquent $\Delta, \neg B \vdash B$, alors qu'au départ on devait démontrer le séquent $\Gamma \vdash B$. Comme Γ est inclus dans Δ, on a plus d'information et l'on est ainsi mieux armé pour la démonstration.

Démontrer, à l'aide de cet axiome (et de tactiques si vous le souhaitez), les propositions suivantes:

1. $(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$
2. $\neg\neg A \Rightarrow A$
3. $\vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (la "loi de Peirce")
4. $\neg(\forall x P(x)) \Rightarrow \exists x(\neg P(x))$
5. $\exists y \forall x(P(y) \Rightarrow P(x))$ (le "paradoxe du buveur").

5 Ecrire des programmes en Coq

Il faut d'abord savoir définir son propre type de données. On utilise en général les types de données inductifs. Par exemple, pour les listes:

```
Inductive my_list: Set :=
  | nil: my_list
  | cons: nat -> my_list -> my_list.
```

Cela correspond exactement aux déclarations de type d'un langage de programmation fonctionnel avec typage fort, tel que OCaml ou Haskell.

1. définir la liste `[1;2;3]`, appeler-la `trois`.

On peut maintenant définir une fonction qui retourne la tête d'une liste, par *filtrage* ou, en anglais, pattern matching:

```
Definition head (l:my_list) : nat :=
  match l with nil => 0
    | cons t q => t
  end.
```

2. définir sur le même modèle une fonction `tail` qui retourne la *queue* d'une liste.

On peut maintenant prouver certaines assertions sur notre code, par exemple:

```
Theorem tete_test_vaut_1 : (head trois) = 1.
```

3. Prouver ce théorème. Pour cette preuve, qui est (presque) triviale, il est utile de "déplier" les définition de `head` et de `test`, le reste se fera automatiquement (on pourra utiliser la tactique `auto`, ou bien se référer à la librairie standard de Coq).

Voici maintenant comment définir la fonction longueur:

```
Fixpoint length (l:ma_liste) : nat :=
  match l with nil => 0
    | cons m l1 => S (length l1)
  end.
```

C'est une définition *récursive*, d'où le mot-clef `Fixpoint`. Notez que Coq vérifie scrupuleusement que votre appel récursif se fait selon les règles de l'art: un argument récursif décroissant structurellement (`l1` est une liste strictement plus petite que `l`) et une analyse de cas *complète*. Le premier point en particulier, peut parfois être limitant et/ou difficile à gérer, contrairement à ce qui peut se passer dans des langages de programmation "normaux".

4. Montrer les théorèmes suivants:

   ```
   Theorem length_1: length test = 3.
   Theorem length_2: length (tail test) = 2.
   ```

5. plus difficile: montrer que si une liste est non vide, alors sa longueur est égale à la longueur de sa queue plus un. A ce point du TP, il pourra être utile de
   - faire des *distinctions de cas*.
   - utiliser des théorèmes de la librairie standard de Coq: on doit inclure à la main les fichiers, avec la commande `Require Import Arith.` (si on a besoin des théorèmes et définitions situés dans `Arith`).
   - utiliser des tactiques de plus haut niveau (telles que `auto`, `eauto`, voire `rewrite`, `omega` ou `ring`) histoire de ne pas passer à chaque fois trois heures à montrer que `1` est différent de `0` ou à remplacer `2+2` par `4`.
6. s'il vous reste du temps, vous pouvez définir les fonctions usuelles d'inversion de liste (indication : commencez par définir une fonction à deux arguments un peu plus générale), de concaténation, etc, et prouver les propriétés que vous voudrez sur ces listes. Par exemple:
   - la longueur de l'inverse d'une liste est égale à la longeur de cette même liste (difficile) ;
   - l'inverse de l'inverse d'une liste est égal cette même liste (difficile) ;
   - la longueur de la concaténation de deux liste est égale à la somme des longeurs des deux listes (facile).

6 Extraire des programmes à partir de preuves

Nous allons enfin brièvement parler de l'*extraction* d'un programme à partir d'une preuve. Il s'agit de celui qui exploite de la manière la plus frappante la correspondance de Curry-Howard. En effet:

- considérons une preuve d'une proposition du type "il existe x tel que BLABLA". Cette preuve *trouve* le témoin `t` et *prouve* BLABLA.
- d'un point de vue strictement informatique, la preuve contient, entre autres, un *algorithme* qui construit `t`: tout est décrit dans la preuve.
- on doit donc être capable d'extraire un programme correspondant à cette preuve.

Le terme de preuve associé à la règle $\exists$-intro est une paire:

$$\exists\text{-intro}\ \frac{\Gamma \vdash \pi : \{t/x\}A}{\Gamma \vdash \langle t, \pi\rangle : \exists x A}$$

Une preuve d'une propriété existentielle *contient* donc, en substance, un témoin t. Celui-ci peut cependant être caché: une preuve du séquent $\Gamma \vdash \exists x A$ ne commence pas forcément par une règle $\exists$-intro. En particulier, il peut y avoir des coupures, et c'est l'élimination des coupures (ou encore, à travers la correspondance, le calcul effectué par le λ-terme lorsqu'il se β-réduit) qui *calcule effectivement* ce terme.

Bien évidemment, si dans Coq il est indispensable d'avoir une *preuve* que le témoin `t` respecte la propriété BLABLA, dans un langage de programmation normal, la preuve en soi peut disparaître, pour ne laisser place qu'à l'information effective permettant de calculer `t`.

6.1 Exemples

Un exemple trivial est la preuve du théorème suivant:

```
Lemma ziro: {m: nat | m=0}.
exists 0.
auto.
Qed.
Recursive Extraction ziro.
```

Ici, on *doit* utiliser la construction `{m: nat | m=0}` en lieu et place de la construction `exists m: nat, m=0`. Il y a une raison théorique[7] qui fait que pour effectuer l'extraction nous devons travailler, non pas dans `Prop` mais dans `Set` (ou dans `Type`), et c'est la manière d'écrire le quantificateur existentiel dans ces types.

Pour revenir à notre exemple, nous extrayons une fonction constante à zéro variables, ce qui correspond exactement à la preuve.

Un exemple un peu plus intéressant est celui-ci:

```
Theorem minus_2: forall n: nat, n > 0 -> { m: nat | m < n}.
```

Faites-en la preuve, puis extrayez-en un programme. Vous aurez au moins besoin de la librairie `Lt`, voire de toute la librairie `Arith`, ainsi qu'une anaylse de cas sur `n` (avec `case`). application of the induction principle).

Questions supplémentaires:

1. pour l'instant, l'extraction extrait aussi la notion de nombre entier, or nous souhaiterions réutiliser les nombres entiers de OCaml (le type `int`). Comment modifier l'extraction pour ce faire ?
2. essayez de prouver des théorèmes plus intéressants d'un point de vue calculatoire, par exemple la division euclidienne. Attention, il faudra raisonner par induction/récurrence.
3. vous pouvez aussi jouer à coqoban

6.2 Preuve et spécifications

S'il est possible d'extraire un programme à partir d'une preuve d'une certaine formule, c'est que cette formule décrit *exactement* ce que le programme doit faire. On sait donc, puisque la preuve a été vérifiée (et peut être revérifiée par d'autres assistants à la preuve), que le programme respecte exactement ses spécifications. Dans l'exemple précédent, on produira donc un programme qui prend un entier naturel différent de zéro et qui renvoie un autre entier naturel plus petit. Notons que nous aurions pu faire la preuve de différentes manières (en choisissant `0` comme témoin, ou bien `n-3`, etc) et que cela aurait alors aboutit à des programmes différents.

L'opération inverse est aussi utilisée: il s'agit, à partir d'un programme préexsitant, de le spécifier suffisamment finement (en particulier au niveau des conditions sur les entrées du programme), pour être capable de le transformer en *preuve* de sa propre spécification.

7 Pour se documenter

Un livre complet sur Coq: le livre Coq'Art (http://www.labri.fr/perso/casteran/CoqArt/). Les exercices du livre sont disponibles ici: http://www.labri.fr/perso/casteran/CoqArt/contents.html

En particulier, pour des preuves purement "logiques", on pourra se référer aux exercices suivants: http://www.labri.fr/perso/casteran/C

La documentation complète de Coq est disponible sur le site http://coq.inria.fr. On pourra en particulier se référer à la FAQ de Coq: https://github.com/coq/coq/wiki/The-Coq-FAQ ou bien au tutorial: https://coq.inria.fr/tutorial/ on encore à un autre tutorial: https://coq.inria.fr/tutorial-nahas

Une référence rapide sur les différentes tactiques en Coq: https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html

[7] Le type `Prop` ne possède que la seconde projection de la paire $\langle t, A\rangle$ associée à la règle $\exists$-intro, car c'est un type imprédicatif et avoir la possibilité d'extraire la première composante de la paire rendrait le calcul des constructions incohérent.