Initiation aux preuves formelles : Partiel écrit

17 décembre 2024 — Sergueï Lenglet et Pierre Rousselin

Consignes et conseils :

- Durée 2h00. Un résumé de cours manuscrit est autorisé.
- Le barème est donné à titre indicatif et est susceptible d'être modifié. Le total des points avant modification éventuelle du barème est 32. La note sera ramenée sur 20.
- Les exercices sont indépendants les uns des autres.
- La clarté des réponses et la propreté de la copie seront des éléments importants de l'évaluation.
- En annexe sont rappelés certains lemmes qui peuvent être utilisés dans certains exercices.

Exercice 1 : Effet des tactiques sur un état de preuve (8 points)

1. On rappelle les 4 résultats suivants sur les nombres réels :

```
add_eq_compat_l : forall r r1 r2 : R, r1 = r2 -> r + r1 = r + r2
add_eq_compat_r : forall r r1 r2 : R, r1 = r2 -> r1 + r = r2 + r
add_eq_reg_l : forall r r1 r2 : R, r + r1 = r + r2 -> r1 = r2
add_eq_reg_r : forall r r1 r2 : R, r1 + r = r2 + r -> r1 = r2
```

Dans chacun des états de preuve suivants, dire comment est modifié l'état de la preuve par la tactique donnée, ou bien si la tactique provoque une erreur, auquel cas en donner la raison.

a)
```
x, y : R
H : x + x = y + y
======================== (1 / 1)
x = y
```
```
apply (add_eq_reg_l x).
```

b)
```
x, y : R
H : x + x = y + y
======================== (1 / 1)
x = y
```
```
apply add_eq_reg_l.
```

c)
```
x, y : R
H : x = y
======================== (1 / 1)
x + y = x + x
```
```
rewrite (add_eq_reg_l x y x).
```

d)
```
x : R
H : x + x = x + 0
======================== (1 / 1)
x = 0
```
```
apply add_eq_reg_l in H.
```

2. On travaille maintenant sur les entiers de type `nat`, et les tactiques `simpl` et `discriminate`. On rappelle la définition de la multiplication des entiers naturels :

```
Fixpoint mul n m =
  match n with
  | 0 => 0
  | S p => m + mul p m
  end.
```

Dans chacun des cas suivants dire l'effet de la tactique donnée sur l'état de preuve donné. En cas d'erreur, expliquer cette erreur.

a)
```
n : nat
H : n = 21
======================== (1 / 1)
2 * n = 42
```

```
simpl.
```

b)
```
n : nat
H : n = 21
======================== (1 / 1)
n * 2 = 42
simpl.
```

c)
```
n : nat
H : 1 = S (S n)
======================== (1 / 1)
2 + 2 = 5
discriminate H.
```

d)
```
n : nat
H : 0 * n = 0
======================== (1 / 1)
2 = 2
discriminate H.
```

--- * ---

Correction de l'exercice 1 :

1. a) Le type de `(add_eq_reg_l x)` est `forall r1 r2 : R, x + r1 = x + r2 -> r1 = r2`. Avec `apply`, le but est unifié avec la conclusion `r1 = r2`, donc `r1` prend la valeur `x` et `r2` prend la valeur `y`. C'est donc l'implication `x + x = x + y -> x = y` qui est appliquée au but, lequel devient `x + x = x + y`.

b) Avec `apply add_eq_reg_l`, Coq cherche à unifier `r1 = r2` avec `x = y`, donc `r1` devrait prendre la valeur `x` et `r2` la valeur `y` mais il n'y a aucun moyen de deviner la valeur de `r`. La tactique échoue donc.

c) Avec `rewrite (add_eq_reg_l x y x)`, le type de la règle de réécriture est `x + y = x + x -> y = x`, Coq va donc changer `y` en `x` dans le but et créer un sous-but pour la prémisse, qui est `x + y = x + x`.

d) Cette fois l'unification a lieu dans l'hypothèse `H` avec la prémisse de l'implication `forall r r1 r2, r + r1 = r + r2 -> r1 = r2`. On unifie donc `r + r1 = r + r2` avec `x + x = y + y`, ce qui fait que `r1` devrait prendre la valeur `x`, `r2` la valeur `y` mais `r` devrait prendre à la fois la valeur `x` et la valeur `y`, ce qui n'est pas possible. La tactique échoue donc.

2. a) La multiplication est définie par *pattern matching* sur son opérande de gauche. Ici, on a `2 * n = S (S 0) * n = n + (S 0) * n = n + (n + 0 * n) = n + (n + 0)` Donc `simpl` réduit le but en `n + (n + 0) = 42`

b) En revanche, `simpl` n'a aucun effet sur ce but, aucune règle de réduction ne s'applique.

c) `discriminate H` permet de passer de $S0 = S(Sn)$ à $0 = Sn$ qui est contradictoire, la preuve est terminée (on a prouvé `False`).

d) `discriminate H` va réduire `H` en $0 = 0$, puis provoquer une erreur (*not a discriminable equality*) car l'égalité n'est clairement pas contradictoire.

--- * ---

Exercice 2 : Prouvable ou non ? (8 points)

Pour chacun des états de preuve suivants, dire s'ils sont prouvables ou non. Dans le cas où il sont prouvables donner une preuve en Coq, dans le cas où ils ne sont pas prouvable, donner un contre-exemple, c'est-à-dire des propositions ou des valeurs de variables rendant vrai le contexte mais pas le but. Pour prouver les lemmes prouvables, vous pouvez utiliser n'importe quel lemme donné à la fin du sujet.

1.
```
A, B, C, D : Prop
H1 : A \/ B
H2 : A -> C
H3 : B -> D
H4 : C -> D
======================== (1 / 1)
D
```

2.
```
======================== (1 / 1)
exists n : nat, n + n = n
```

3.
```
========================== (1 / 1)
forall n : nat, exists p : nat, n = S p
```

4. Pour cette question, on pourra utiliser l'axiome du tiers exclu, que voici :

```
classic : forall P : Prop, P \/ ~ P
```

Rappel : en Coq, si `P` est une proposition, `~P` est une notation pour `P -> False`.

```
P : Prop
H : ~ ~ P
========================== (1 / 1)
P
```

--- * ---

Correction de l'exercice 2 :

1. C'est prouvable en Coq.
```
destruct H1 as [H1 | H1].
- apply H4. apply H2. exact H1.
- apply H3. exact H1.
```

2. C'est prouvable en Coq.
```
exists 0.
rewrite add_0_l. (* ou [simpl] ou déjà [reflexivity]. *)
reflexivity.
```

3. C'est faux. On peut prendre $n = 0$. S'il existe un `p` tel que $0 = Sp$, on peut **`discriminate`** cette égalité pour prouver `False`.

4. C'est prouvable avec le tiers exclu.
```
destruct (classic P) as [H1 | H1].
- exact H1.
- exfalso. apply H. exact H1.
```

--- * ---

Exercice 3 : Ordre sur les réels et preuve mathématique détaillée (8 points)

1. On démontre ici le théorème suivant : soit f une fonction affine strictement croissante, alors son coefficient directeur est strictement positif. On suppose connus seulement les résultats suivants :
```
lt_0_1 : 0 < 1
mul_0_l : forall r : R, 0 * r = 0
mul_0_r : forall r : R, r * 0 = 0
mul_1_l : forall r : R, 1 * r = r
mul_1_r : forall r : R, r * 1 = r
mul_lt_compat_l : forall r r1 r2 : R, 0 < r -> r1 < r2 -> r * r1 < r * r2
mul_lt_compat_r : forall r r1 r2 : R, 0 < r -> r1 < r2 -> r1 * r < r2 * r
mul_lt_reg_l : forall r r1 r2 : R, 0 < r -> r * r1 < r * r2 -> r1 < r2
mul_lt_reg_r : forall r r1 r2 : R, 0 < r -> r1 * r < r2 * r -> r1 < r2
add_0_l : forall r : R, 0 + r = r
add_0_r : forall r : R, r + 0 = r
add_comm : forall r1 r2, r1 + r2 = r2 + r1
add_lt_compat_l : forall r r1 r2, r1 < r2 -> r + r1 < r + r2
add_lt_compat_r : forall r r1 r2, r1 < r2 -> r1 + r < r2 + r
add_lt_reg_l : forall r r1 r2 : R, r + r1 < r + r2 -> r1 < r2
add_lt_reg_r : forall r r1 r2 : R, r1 + r < r2 + r -> r1 < r2
```

Recopier sur la copie et annoter la preuve suivante pour y faire apparaître les applications, réécritures (ou définitions) permettant de justifier chaque étape.

Démonstration. Soient a et b tels que pour tout réel x, $f(x) = ax + b$ (hypothèse (E)). Comme f est supposée croissante, on a, pour tous réels x et y, $x < y \implies f(x) < f(y)$ (hypothèse (H)). On sait que $0 < 1$. Or,

$$\begin{array}{c} 0 < 1 \\ \Downarrow \\ f(0) < f(1) \\ \Downarrow \\ a \times 0 + b < a \times 1 + b \\ \Downarrow \\ a \times 0 < a \times 1 \\ \Downarrow \\ 0 < a \times 1 \\ \Downarrow \\ 0 < a \end{array}$$

□

2. On suppose connues par Coq les définitions suivantes :

```
Definition increasing (f : R -> R) := forall (x y : R), x < y -> f x < f y.
Definition aff (a b : R) (x : R) := a * x + b.
```

En résumé, `(aff a b)` est la fonction affine qui à x associe $ax+b$. Écrire la preuve en Coq du lemme suivant :

```
Lemma aff_incr_a_pos (a b : R) : increasing (aff a b) -> 0 < a.
```

3. Écrire la preuve mathématique très détaillée du résultat suivant : toute fonction affine de coefficient directeur strictement positif est strictement croissante.

4. Écrire la preuve en Coq du lemme :

```
Lemma a_pos_aff_incr (a b : R) : 0 < a -> increasing (aff a b).
```

--- * ---

Correction de l'exercice 3 :

1. La première implication est prouvée en appliquant l'hypothèse (H) (la croissance stricte de f) à la première inégalité. La suivante en réécrivant avec l'hypothèse (E). La troisième en appliquant le lemme `add_lt_reg_r b (a * 0) (a * 1)`. Les deux dernières en réécrivant avec respectivement `(mul_0_r a)` et `(mul_1_r a)`

2.
```
Lemma aff_incr_a_pos (a b : R) : increasing (aff a b) -> 0 < a.
Proof.
  intros H.
  specialize (H 0 1 lt_0_1).
  unfold aff in H.
  rewrite mul_0_r, mul_1_r in H.
  apply add_lt_reg_r in H.
  exact H.
Qed.
```

3. Soit f une fonction affine. Soient a et b deux réels tels que pour tout réel x, $f(x) = ax + b$. On

suppose que $a > 0$. Soient x et y deux réels tels que $x < y$. On a

$$\begin{array}{rl} x < y & \\ \Downarrow & \\ ax < ay & \text{(en appliquant } \texttt{(mul\_lt\_compat\_l a)}\text{, car } a > 0) \\ \Downarrow & \\ ax + b < ax + b & \text{(en appliquant } \texttt{(add\_lt\_compat\_r b)}\text{)} \\ \Downarrow & \\ f(x) < f(y) & \end{array}$$

4.
```
Lemma a_pos_aff_incr (a b : R) : 0 < a -> increasing (aff a b).
Proof.
  unfold aff.
  intros Ha x y Hxy.
  apply add_lt_compat_r.
  apply mul_lt_compat_l.
  - exact Ha.
  - exact Hxy.
Qed.
```

--- * ---

Exercice 4 : Opération `ordre` sur `nat`, à valeurs dans un type à trois éléments (8 points)

Dans cet exercice, on se donne le type `Comparaison` qui est un type inductif à 3 éléments :

```
Inductive Comparaison := Lt | Eq | Gt.
```

Intuitivement, `Lt` signifie « *less than* », (strictement) inférieur, `Eq` signifie « *equal* », égal, et `Gt` signifie « *greater than* », (strictement) supérieur. On définit, d'une certaine manière, la fonction `ordre : nat -> Comparaison` dont on connaît les 4 propriétés suivantes :

```
ordre_0_0: ordre 0 0 = Eq
ordre_0_succ: forall m : nat, ordre 0 (S m) = Lt
ordre_succ_0: forall n : nat, ordre (S n) 0 = Gt
ordre_succ_succ: forall n m : nat, ordre (S n) (S m) = ordre n m
```

1. En utilisant ces 4 règles, évaluer `ordre 5 2`, `ordre 1 3` et `ordre 2 2`.

2. Prouver en Coq le lemme suivant :
```
Lemma ordre_diag (n : nat) : ordre n n = Eq.
```

3. Prouver en Coq le lemme suivant :
```
Lemma ordre_succ_diag_r (n : nat) : ordre n (S n) = Lt.
```

4. On **admet** le résultat suivant :
```
ordre_Lt_trans: forall (n m p : nat), ordre n m = Lt -> ordre m p = Lt -> ordre n p = Lt.
```
Prouver en Coq le lemme suivant :
```
Lemma ordre_add_r (n p : nat) : ordre n ((S p) + n) = Lt.
```

--- * ---

Correction de l'exercice 4 :

1. a)

$$\texttt{ordre}(5,2) \overset{\texttt{ordre\_succ\_succ}}{=} \texttt{ordre}(4,1) \overset{\texttt{ordre\_succ\_succ}}{=} \texttt{ordre}(3,0) \overset{\texttt{ordre\_0\_succ}}{=} \texttt{Gt}.$$

b)

$$\texttt{ordre}(1,3) \overset{\texttt{ordre\_succ\_succ}}{=} \texttt{ordre}(0,2) \overset{\texttt{ordre\_succ\_0}}{=} \texttt{Lt}.$$

c)

$$\texttt{ordre}(2,2) \overset{\texttt{ordre\_succ\_succ}}{=\!=} \texttt{ordre}(1,1) \overset{\texttt{ordre\_succ\_succ}}{=\!=} \texttt{ordre}(0,0) \overset{\texttt{ordre\_0\_0}}{=\!=} \texttt{Eq}.$$

2.
```
Lemma ordre_diag (n : nat) : ordre n n = Eq.
Proof.
  induction n as [|n IH].
  - rewrite ordre_0_0. reflexivity.
  - rewrite ordre_succ_succ. exact IH.
Qed.
```

3.
```
Lemma ordre_succ_diag_r (n : nat) : ordre n (S n) = Lt.
Proof.
  induction n.
  - rewrite ordre_0_succ. reflexivity.
  - rewrite ordre_succ_succ. rewrite IHn. reflexivity.
Qed.
```

4.
```
Lemma ordre_add_r (n p : nat) : ordre n ((S p) + n) = Lt.
Proof.
  induction p.
  - rewrite add_1_l. exact (ordre_succ_diag_r n).
  - rewrite add_succ_l. destruct n as [| n].
    * rewrite ordre_0_succ. reflexivity.
    * rewrite ordre_succ_succ. apply (ordre_Lt_trans _ (S n)).
      + exact (ordre_succ_diag_r n).
      + exact IHp.
Qed.
```

--- * ---

Liste de lemmes qu'on peut utiliser

```
add_0_l : forall n : nat, 0 + n = n
add_0_r : forall n : nat, n + 0 = n
add_1_l : forall n : nat, 1 + n = S n
add_1_r : forall n : nat, n + 1 = S n
add_assoc : forall n m p : nat, (n + m) + p = n + (m + p)
add_comm : forall n m : nat, n + m = m + n
add_succ_l : forall n m : nat, S n + m = S (n + m)
add_succ_r : forall n m : nat, n + S m = S (n + m)
lt_0_1 : 0 < 1
mul_0_l : forall r : R, 0 * r = 0
mul_0_r : forall r : R, r * 0 = 0
mul_1_l : forall r : R, 1 * r = r
mul_1_r : forall r : R, r * 1 = r
mul_lt_compat_l : forall r r1 r2 : R, 0 < r -> r1 < r2 -> r * r1 < r * r2
mul_lt_compat_r : forall r r1 r2 : R, 0 < r -> r1 < r2 -> r1 * r < r2 * r
mul_lt_reg_l : forall r r1 r2 : R, 0 < r -> r * r1 < r * r2 -> r1 < r2
mul_lt_reg_r : forall r r1 r2 : R, 0 < r -> r1 * r < r2 * r -> r1 < r2
add_0_l : forall r : R, 0 + r = r
add_0_r : forall r : R, r + 0 = r
add_comm : forall r1 r2, r1 + r2 = r2 + r1
add_lt_compat_l : forall r r1 r2, r1 < r2 -> r + r1 < r + r2
add_lt_compat_r : forall r r1 r2, r1 < r2 -> r1 + r < r2 + r
add_lt_reg_l : forall r r1 r2 : R, r + r1 < r + r2 -> r1 < r2
add_lt_reg_r : forall r r1 r2 : R, r1 + r < r2 + r -> r1 < r2
```