INITIATION AUX PREUVES FORMELLES : TD 2
septembre 2024 — Pierre Rousselin

La tactique `rewrite` permet de remplacer un terme par un autre dans le but. Pour ce faire, elle utilise l'arbre *syntactique* du but et le parcourt *en profondeur d'abord*, en commençant par la gauche.

Une fois qu'elle a trouvé un sous-terme `t` qui *correspond* à la règle de réécriture donnée, tous les sous-termes identiques à `t` sont réécrits. Dans ce cas, on peut préciser les numéros des occurrences (dans l'ordre du parcourt) que l'on veut remplacer.

Exercice 1 : Réécritures simples (sans unification) et arbre syntactique

Pour chacun des états de preuve suivant, représenter l'arbre syntactique du but, puis dire quel effet ont les tactiques données dans l'énoncé.

<p>1. <code>n : nat</code>
 <code>H : n = 2</code>
 ===== (1 / 1)
 <code>n + n = 4</code>

 <code>rewrite H.</code></p> | <p>3. <code>n : nat</code>
 <code>H : n = 2</code>
 ===== (1 / 1)
 <code>n + n = 2 + 2</code>

 <code>rewrite <-H.</code></p> |
| <p>2. <code>n : nat</code>
 <code>H : n = 2</code>
 ===== (1 / 1)
 <code>n + n = 4</code>

 <code>rewrite H at 2.</code></p> | <p>4. <code>n : nat</code>
 <code>H : n = 2</code>
 ===== (1 / 1)
 <code>n + n = 2 + 2</code>

 <code>rewrite <-H at 1.</code>
 <code>rewrite H at 2.</code></p> |
- * ---

Exercice 2 : Instancier des théorèmes

Ci-dessous sont donnés des contextes, puis un terme. Quel est son type? Terminer la preuve en utilisant le terme donné ou bien un terme légèrement modifié.

1. `n, m : nat`
`add_comm : forall n m : nat, n + m = m + n`
===== (1 / 1)
`m * 2 + n = n + m * 2`
Terme à typer : `(add_comm (m * 2) n)`
 2. `n, m : nat`
`add_assoc : forall n m p : nat, n + (m + p) = n + m + p`
===== (1 / 1)
`42 + n + 3 = 42 + (n + 3)`
Terme à typer : `(add_assoc 42)`.
 3. `p, q, r : nat`
`add_eq_reg_r : forall n m p : nat, n + p = m + p -> n = m`
`E : p + r * r = q + r * r`
===== (1 / 1)
`p = q`
Terme à typer : `(add_eq_reg_r p q (r * r))`
 4. Même contexte, typer `(add_eq_reg_r p q (r * r) E)`. Que dire du terme `(add_eq_reg_r _ _ _ E)` s'il doit s'appliquer au but courant?
- * ---

Exercice 3 : `rewrite` avec instanciation et/ou unification

On rappelle les règles de réécriture couramment utilisées sur l'addition sur `nat`.

```

add_0_r: forall n : nat, n + 0 = n.
add_0_l: forall n : nat, 0 + n = n.
add_succ_r: forall n m : nat, n + S m = S (n + m).
add_succ_l: forall n m : nat, S n + m = S (n + m).
add_succ_comm : forall n m : nat, n + S m = S n + m
add_comm: forall n m : nat, n + m = m + n.
add_assoc: forall n m p : nat, n + (m + p) = n + m + p.

```

Dans chacun des cas suivants, donner l'effet de la tactique donnée, ou bien, s'il y a une erreur, dire laquelle. On dessinera l'arbre syntaxique associé au but.

1. `n : nat`
===== (1 / 1)
`n + 0 + 1 = 1 + 0 + n`

`rewrite add_0_r.`

2. `n : nat`
===== (1 / 1)
`n + 0 + 1 = 1 + (n + 0)`

`rewrite add_0_r.`

3. `n, m, p : nat`
===== (1 / 1)
`n + m + p = p + m + n`

`rewrite add_comm.`

4. `n, m, p : nat`
===== (1 / 1)
`n + m + p = p + m + n`

`rewrite (add_comm n).`

9. Résoudre le but suivant (tiré de la preuve de `mul_succ_r`) :

```

n', m : nat
===== (1 / 1)
n' * m + m + S n' = n' * m + m + S n'

```

--- * ---

Exercice 4 : `apply` versus `rewrite`

Il arrive souvent que des termes soient des implications (quantifiées ou non) dont le but est une égalité. Dans ce cas, l'utilisateur peut `apply` ce terme (comme une implication) ou `rewrite` avec ce terme (auquel cas, Coq demandera une preuve des prémisses).

Dans chacun des cas suivant, dire l'effet de `apply H` ou `rewrite H`. Si il manque des arguments que Coq ne peut pas deviner, dire lesquelles et terminer la preuve.

1. `p, q, r : nat`
`H : forall n m p : nat, n = m -> n + p = m + p`
`E : p = q`
===== (1 / 1)
`p + r * r = q + r * r`

2. `p, q, r : nat`
`H : forall n m p : nat, n + p = m + p -> n = m`
`E : p + r * r = q + r * r`
===== (1 / 1)
`p = q`

--- * ---

5. `n, m, p : nat`
===== (1 / 1)
`n + m + p = p + m + n`

`rewrite add_assoc.`

6. `n, m, p : nat`
===== (1 / 1)
`n + m + p = p + m + n`

`rewrite <-add_assoc.`

7. `n, m, p : nat`
===== (1 / 1)
`n + m + p = p + m + n`

`rewrite <-(add_assoc p).`

8. Résoudre le but suivant à l'aide des règles de réécritures données plus tôt.

```

n, m, p : nat
===== (1 / 1)
n + m + p = p + m + n

```