Présentation de la correspondance de Curry-Howard

F. BEN MENA

[2022-05-11 mer.]

Table des matières

1 Introduction

«The proof of the pudding is in the eating.»

— Proverbe anglais.

1.1 La barrière d'entrée

La raison d'être de cette présentation est double. Il s'agit de montrer que les frontières que l'on pense étanches entre mathématique et informatique forment une représentation impropre du payasage dans lequel ces deux disciplines évoluent. Les échanges mutuels entre elles sont évidents depuis les travaux fondateurs de Kurt Gödel, Alonzo Church, Gerhard Gentzen et Alan Turing. Ces travaux d'une synchronicité remarquable sont publiés, indépendamment les uns des autres, en un laps de temps de moins d'une décennie (autour des années 1930). Depuis, ils ont été poursuivis, enrichis et ont conduit à quelques résultats remarquables. L'un d'entre eux est aujourd'hui connu sous le nom de correspondance de Curry-Howard. Aborder ce résultat lorsqu'on a une formation en «mathématiques fondamentales» a quelque chose de déroutant. Nos représentations, nous amènent à penser, et parfois à enseigner, qu'il existe une dichotomie nette entre calcul et raisonnement ce que, d'une certaine manière, dément cette correspondance de Curry-Howard.

Mais la deuxième raison d'être de cette présentation n'est pas épistémologique. Sans même aborder la démonstration formelle du principal résultat théorique, elle va se limiter à donner un aperçu de ses conséquences. Avec le souci d'abaisser la barrière d'entrée dans un cadre théorique différent de celui bien connu de la théorie des ensembles. Nous allons faire des mathématiques et de l'informatique — on ne verra pas de frontière tangible — sur des objets que nous connaissons bien.

1.2 La notion de type

Considérons la notion de *type* en informatique. Elle est conçue pour permettre d'organiser et de classer les données. Chaque langage de programmation présente divers types prédéfinis ainsi que des moyens d'étendre les types existants. Partons d'un problème parfaitement banal et construisons en *Coq* un type destiné à travailler avec les jours de la semaine. Voici ce que l'on peut écrire :

```
1  Inductive jour_semaine : Set :=
2    | Dimanche
3    | Lundi
4    | Mardi
5    | Mercredi
6    | Jeudi
7    | Vendredi
8    | Samedi.
9
10 Definition jour_suivant (jour : jour_semaine) : jour_semaine := match jour with
11   | Dimanche => Lundi
12   | Lundi => Mardi
13   | Mardi => Mercredi
14   | Mercredi => Jeudi
15   | Jeudi => Vendredi
16   | Vendredi => Samedi
17   | Samedi => Dimanche
18 end.
19
20 Compute (jour_suivant Dimanche).
```

Avec cet extrait nous disposons :

— d'un nouveau type de données, nommé `jour_semaine` ;
— d'une fonction qui permet le calcul du jour qui succède à un jour donné ;
— d'un moyen d'appeler la fonction précédente sur une valeur du type idoine.

Dès lors que l'on peut construire des types de données, des fonctions opérant sur les valeurs de ce type et que l'on peut appeler ces fonctions, on se dit que ce logiciel présente certaines caractéristiques d'un langage de programmation. *Coq* est bien un langage de programmation, mais il n'est pas que cela.

Considérons le code suivant :

```
Definition est_premier (n : nat) : Prop :=
  ∀ (n1 n2 : nat), n = n1 * n2 -> (n1 = 1 /\ n2 = n) \/ (n1 = n /\ n2 = 1).
```

où les symboles $\wedge, \vee$ représentent les connecteurs propositionnels de la conjonction et de la disjonction. Ce qu'illustre cet exemple, c'est que ce langage est assez expressif pour traduire un très vaste domaine des énoncés mathématiques. Comme l'indique la partie comprise entre l'identifiant et le symbole `:=`, le type de cette fonction est `nat -> Prop`, autrement dit, il s'agit d'un *prédicat* sur les entiers.

2 Un survol de *Coq*

Sans entrer dans les détails de la programmation fonctionnelle, on va en indiquer les points saillants.

2.1 De la nature des types de données

Introduisons des types de données simples. Les types de données sont des éléments qui relèvent de *constructions inductives*. Contrairement à la majorité des langages de programmation, les types de données en *Coq* ne sont pas transparents. Dès lors, il est non seulement loisible de se pencher sur la construction de ces types mais il est même presque indispensable de le faire.

Le type booléen comporte deux *constructeurs* et deux valeurs. Le type des entiers naturels comporte deux *constructeurs* et une infinité de valeurs.

```
Inductive bool : Set :=
  | true
  | false.

Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

On reconnait dans le type des entiers naturel la construction due à Peano. Un entier naturel est soit 0 soit le *successeur* d'un autre entier. Le constructeur s est cette fonction successeur. Toutes les fonctions qui opèrent sur les valeurs de l'un de ces deux types prennent appui ces constructions et sur rien d'autre. Avant d'aborder des exemples de fonctions sur sur les entiers et les booléens, abordons deux types supplémentaires dont il sera question plus loin :

```
Inductive dessert : Set := pudding.
Inductive vide : Set := .
```

Le type `dessert` admet un unique constructeur qui est `pudding`, le type `vide` n'admet aucun constructeur.

Cependant, ce dernier type est parfaitement construit, bien qu'aucune valeur ne puisse être crée avec ce type.

2.1.1 La stratification des types

Dans les exemples introductifs, que ce soit pour la construction `jour_semaine` ou `est_premier` on a rencontré les mots-clés `Set` et `Prop`. Le premier défini la *sorte des spécifications* et le second la *sorte des propositions*. Dans *Coq* le langage de formation des termes et des types s'appelle *Gallina*.

considérons le type `nat` des entiers naturels. Il s'agit également d'un terme dont le type est `Set`. De la même manière, lorsque `P` une variable propositionnelle, son type est `Prop`. Dans le langage *Gallina*, en tant que termes, `Set` et `Prop` ont un type qui est `Type`. À son tour, le terme `Type` admet un type qui est `Type`. Cette situation n'est pas sans rappeler le paradoxe de la théorie des ensembles lorsque l'on tente de définir l'ensemble de tous les ensembles (paradoxe attribué à B. Russel). En réalité, il existe une hiérarchie de types bien que l'index propre à chaque niveau hiérarchique ne soit pas explicitement affiché. Ainsi `Set` et `Prop` sont des termes dont le type a pour index `Type@2`. Le type ce dernier terme a pour index `Type@3`, *etc*.

Revenons aux exemples précités, et explicitons le type de certains termes. La fonction `jour_suivant` est du type `jour_semaine -> jour_semaine` et `jour_semaine` est du type `Set`. Quel est donc le type du terme suivant : `jour_semaine -> jour_semaine` ? La réponse est `Set`.

De même, au sein de la construction de la fonction `est_premier`, les constituants `n = n1 * n2`, `n1 = 1`, ... sont des termes dont le type est `Prop`. Le terme qui désigne l'implication (qui est également le terme quantifié) est également du type `Prop`.

Ainsi les sortes `Set` et `Prop` revêtent un caractère de stabilité pour les fonctions construites à partir de termes typés dans `Set` et `Prop` respectivement. Pour insister sur ce point indiquons que le terme `nat -> nat` est de type `Set` et que le terme `P -> Q`, avec `P Q : Prop`, est de type `Prop`.

2.2 De la nature des fonctions

2.2.1 Fonctions de type flèche

Passons aux fonctions définies sur des termes dont les types sont `nat` ou `bool`. Comme souvent en programmation fonctionnelle, elles sont construites selon un mécanisme de *filtrage* sur les constructeurs. Pour illustrer cela, nous reproduisons le code des fonctions définissant la négation, la conjonction et la disjonction booléennes.

```
1  Definition negation (b : bool) : bool :=
2    match b with
3    | true => false
4    | false => true.
5    end.
6
7  Definition conjonction (b1 b2 : bool) : bool :=
8    match b1, b2 with
9    | true, true => true
10   | _, _ => false
11   end.
12
13 Definition disjonction (b1 b2 : bool) : bool :=
14   match b1, b2 with
15   | false, false => false
16   | _, _ => true
17   end.
```

La négation est une fonction de type `bool -> bool`. La conjonction et la disjonction booléennes sont de type `bool -> bool -> bool`. Le caractère souligné _ signifie que l'on ne prend pas en compte la variable substituée par ce caractère.

L'addition et la multiplication sur les entiers naturels suivent exactement la même idée. La seule différence tient au caractère récursif de chacune de ces fonctions qui impose de remplacer le mot-clé `Definition` par le mot-clé `Fixpoint` par lequel *Coq* s'assure de la terminaison de ces fonctions. Au passage, introduisons des simplifications syntaxiques : 0 remplace le constructeur `O` et lorsque `n` est un entier naturel, son successeur est noté `n.+1` (aucun espace entre l'identificateur et la notation suffixe `.+1`). Le successeur du successeur est noté `n.+2`, *etc.* jusqu'à `n.+5`.

```
Fixpoint addn (n1 n2 : nat) : nat :=
  match n1 with
  | 0 => n2
  | n'.+1 => (addn n' n2).+1
  end.

Fixpoint muln (n1 n2 : nat) : nat :=
  match n1 with
  | 0 => 0
  | n'.+1 => n' + (muln n' n2)
  end.
```

L'addition et la multiplication des entiers sont des fonctions de type `nat -> nat -> nat`.

La notation `(fun a:A => e)` construit expressément une fonction anonyme qui renvoie le terme `e` à partir de l'argument `a:A`. Les notations mathématiques propres à la théorie ensembliste nous font écrire de manière analogue $a \in A \mapsto e$ (l'expression e dépend de la variable a).

De plus, la flèche est associative à droite. Ainsi `A -> B -> C` s'entend comme `A -> (B -> C)`. Dès lors, cette fonction qui prend deux arguments s'identifie à une fonction qui, à partir de la donnée du premier argument, renvoie une fonction. Au niveau de l'application, pour une telle fonction `f`, il n'y pas de différence entre `(f a b)` et `((f a) b)`.

Bien que la notion de produit cartésien de types existe, on privilégie la forme `A -> B -> C` pour la construction d'une fonction à deux arguments à la forme qui opère sur un couple de valeurs, que l'on note habituellement $f : A \times B \longrightarrow C$. La première forme est dite *curryfiée* (sans que cela concerne le domaine des épices...).

Par la suite, sauf mention explicite du contraire, les notations infixes `+` et `*` désigneront l'addition et la multiplication sur les entiers. De même les notations infixes `&&`, `||` désigneront respectivement la conjonction et la disjonction booléennes. La notation préfixe `~~` servira à la négation booléenne. Une autre simplification syntaxique s'applique aux situations où le filtrage est associé à un constructeur par défaut. Ainsi, les fonctions d'addition et de multiplication auraient pu être écrites ainsi :

```
Fixpoint addn (n1 n2 : nat) : nat :=
  if n1 is n'.+1 then (addn n' n2).+1 else n2.

Fixpoint muln (n1 n2 : nat) : nat :=
  if n1 is n'.+1 then n' + (muln n' n2) else 0.
```

2.2.2 Fonctions de type dépendant

Les fonctions en *Coq* n'ont pas nécessairement un type de la forme `A -> B`. Il arrive que le type de la valeur renvoyée par la fonction *dépende de la valeur* de l'argument appliqué à la fonction. Soit `f` une fonction. On considère `B` le type du terme `(f a)`, lequel dépend de la *valeur* de `a` alors, le type de la fonction est noté :

∀ x:A, B. Considérons un exemple simple :

```
Definition PP := fun n => if n is 0 then dessert else nat.

Definition f1 := fun (n : nat) =>
  match n return (PP n) with
  | 0 => pudding
  | _ => n
  end.
```

Pour obtenir le type d'un terme on s'appuie sur la commande `Check`.

```
Check addn.
Check PP.
Check f1.
```

```
: nat -> nat -> nat.
: nat -> Set.
: ∀ n : nat, PP n
```

Si on se reprend les constructions des termes `nat` et `dessert` alors on observe que ces termes sont du type `Set`. Les deux premières fonctions ne sont pas de type dépendant, elles sont du type flèche : `->`. En revanche, la troisième fonction, `f1`, est de type dépendant, elle apparait comme une quantification universelle qui traduit que pour tout entier n, `f1 n` est du type `PP n`.

Pour obtenir le calcul produit par une fonction on s'appuie sur la commande `Compute`.

```
Compute addn 3 5.
Compute PP 0.
Compute PP 4.
Compute f1 0.
Compute f1 1.
```

```
8 : nat
dessert : Set
nat : Set
pudding : PP 0
1 : PP 1
```

Allons plus loin et construisons une fonction dont le type serait `∀ n : nat, PP n -> PP n.+1`. Par exemple :

```
Definition f2 : ∀ n : nat, PP n -> PP n.+1 := fun (n : nat) =>
   match n return (PP n -> PP n.+1) with
   | 0 => fun _ => 1
   | n'.+1 => fun _ => n'.+2
   end.
```

On observe les résultat suivants :

```
Check f2.
Check f2 (n:=3).
Compute f2 (n:=3) 3.
```

```
: ∀ n : nat, PP n -> PP n.+1
: PP 3 -> PP 4
= 4 : PP 4
```

Essayons d'aller encore un peu plus loin et construisons une fonction qui prend deux arguments, l'un du type `PP 0`, l'autre du type `∀ n : nat, PP n -> PP n.+1` et dont le type de la valeur de retour est `∀ n : nat, PP n`.

```
Definition f3 : (PP 0) -> (∀ n : nat, PP n -> PP n.+1) -> ∀ n : nat, PP n :=
  fun (f : PP 0) (ff: ∀ n : nat, PP n -> PP n.+1) =>
    fix F (n : nat) : (PP n) :=
      match n return (PP n) with
      | 0 => f
```

```
        | n'.+1 => ff n' (F n')
        end.
```

Ne sommes-nous pas arriver à quelque chose de bien connu ? En réalité cette fonction, ou sa généralisation, est définie en *Coq* et se nomme `nat_rec`. En effet :

```
Check nat_rec.
```

```
: ∀ P : nat -> Set, P 0 ->
  (∀ n : nat, P n -> P n.+1) ->
  ∀ n : nat, P n
```

Terminons par une dernière construction, quitte à laisser les détails de côté en première lecture :

```
Definition g1 : (∀ n : nat, PP n -> PP n.+1) := fun (n : nat) (m : PP n) =>
  match n return (PP n -> _) with
  | 0 => fun _ => 1
  | n'.+1 => fun m => m + n.+1
  end m.

Definition somme := nat_rec PP pudding g1.
```

On observe les résultats suivants :

```
Check gg.
Check somme.
Compute somme 4.
Compute somme 0.
```

```
: ∀ n : nat, PP n -> PP n.+1
: ∀ n : nat, PP n
= 10 : PP 4
= pudding : PP 0
```

Bien sûr, on pourrait également écrire :

```
Definition somme_bis := f3 pudding g1.
Compute somme_bis 10.
Compute somme_bis 0.
```

```


= 55 : PP 10
= pudding : PP 0
```

Les résultats produits par les fonctions `somme` et `somme_bis` sont identiques. Dans la construction de `f3` on aurait pu introduire une quantification sur `PP`, en position initiale, pour retrouver la structure de `nat_rec`.

2.3 Une approche de Curry-Howard au travers un exemple

La fonction inductive `nat_rec` est produite par *Coq* au moment où le type inductif est construit. Deux autres fonctions similaires sont également produites, il s'agit de `nat_ind` et `nat_rect`. Comparons les types de ces fonctions :

```
nat_rec
     : ∀ P : nat -> Set,
       P 0 -> (∀ n : nat, P n -> P n.+1) -> ∀ n : nat, P n

nat_ind
     : ∀ P : nat -> Prop,
       P 0 -> (∀ n : nat, P n -> P n.+1) -> ∀ n : nat, P n
```

```
nat_rect
     : ∀ P : nat -> Type,
       P 0 -> (∀ n : nat, P n -> P n.+1) -> ∀ n : nat, P n
```

Ce qui distingue les trois fonctions est le type de du premier argument. La fonction `nat_ind` n'est rien d'autre que la **fonction** qui permet d'établir, par récurrence, qu'une propriété est vraie sur les entiers.

Bien que cela soit un peu précoce on peut mettre en œuvre un raisonnement par récurrence, précisément, par appel à la fonction `nat_ind`. Voici l'énoncé qui va être prouvé : `∀ n : nat, n + 0 = n`. Pour alléger l'écriture, nous allons établir un lemme intermediaire (les détails de la preuve de ce lemme peuvent être omis en première lecture).

```
1  Lemma etape_heredite : ∀ n : nat, n + 0 = n -> n.+1 + 0 = n.+1.
2  Proof.
3    move => n H.
4    have Hyp : n.+1 + 0 = (n + 0).+1 => //.
5    rewrite Hyp -[in RHS]H //.
6  Qed.
```

La preuve de notre énoncé consiste en l'appel de la fonction `Check nat_ind` avec les arguments suivants :

— `P := (fun n : nat => n + 0 = n) : nat -> Prop`, `P` est un prédicat sur les entiers ;
— `(erefl 0) : 0 + 0 = 0` est le terme d'initialisation de la récurrence ;
— `etape_heredite : ∀ n : nat, n + 0 = n -> n.+1 + 0 = n.+1` dont le nom est aussi explicite que le type…

Il ne reste plus qu'à finaliser cela :

```
1  Lemma addition_0_neutre_a_droite: ∀ n : nat, n + 0 = n.
2  Proof nat_ind (fun n : nat => n + 0 = n) (erefl 0) etape_heredite.
```

Coq semble satisfait. Le *terme* fourni dans la preuve est non seulement bien typé, le logiciel ne permet pas de circonvenir aux règles de typage, mais en plus le type obtenu est précisément celui qui définit l'énoncé du lemme.

Au travers de cet exemple émergent les deux aspects indissociables de la correspondance de Curry-Howard : Une proposition est un type (*propositions as types*) et la preuve de cette proposition est un programme (*proofs as programs*) dont le type est donné par la proposition.

Il va de soi que la preuve précédente présente deux aspects très discutables :

— d'une part l'écriture du terme `nat_ind` … est quelque peu artificielle ;
— d'autre part, la partie intéressante de la preuve s'est trouvée «externalisée» dans un autre énoncé.

Il reste que *Coq* est également un *assistant de preuve*. Dorénavant, il s'agit de décrire des méthodes intelligibles pour la construction des preuves. Voici comment se déroule la construction de la preuve précédente de manière moins artificielle :

```
1  Lemma lemme_addition_0_neutre_a_droite_bis : ∀ n : nat, n + 0 = n.
2  Proof.
3    elim.
4    by [].
5    move => n HRec.
6    have HypSuppl : n.+1 + 0 = (n + 0).+1.
7      by [].
8    rewrite HypSuppl -[in RHS]HRec.
9    by [].
10 Qed.
```

Les informations utiles étant affichées par le logiciel, on peut les commenter.

Ligne 2. La commande `Proof` bascule dans le mode interactif de preuve. Dès lors, nous sommes en présence d'un *but*. Un but est formé d'un contexte et d'une conclusion. Le contexte regroupe les variables et les hypothèses disponibles alors que la conclusion indique ce qui doit être prouvé. Au début de la preuve, la conclusion du but n'est rien d'autre que l'énoncé.

Ligne 3. La première étape de la preuve consiste en l'invocation de la tactique `elim`. Cette tactique est celle qui met en place la récurrence. Dans le cas présent deux nouveaux buts vont être ouverts. Le premier est pour l'initialisation de la récurrence : son contexte est vide, sa conclusion est `0 + 0 = 0`. Le second but sera mis sur le métier lorsque le but courant sera clos.

Ligne 4. Pour clore, le but courant, on doit saisir un terme dont le type est `0 + 0 = 0`. Or, le terme `erefl 0` admet le type souhaité. On aurait pu écrire `exact: (erefl 0)`. Toutefois, *Coq* est assez aimable pour clore ce but de manière automatisée, ce que suggère le très bref `by []`. *Coq* bascule alors vers le second but dont la conclusion est : `∀ n : nat, n + 0 = n -> n.+1 + 0 = n.+1`.

Ligne 5. On doit *introduire* la variable *n* et l'hypothèse de récurrence `n + 0 = n` dans le contexte de la preuve ce que réalise la commande `move => n HRec`. On observe que la conclusion du but devient `n.+1 + 0 = n.+1`.

Ligne 6. On se dit qu'il serait intéressant d'avoir une hypothèse supplémentaire. On donne lui donne un nom et on écrit son énoncé. Dès lors, le logiciel laisse de côté le but courant, et ouvre un but pour prouver cette hypothèse.

Ligne 7. Notre hypothèse est triviale (on verra pourquoi un peu plus loin). La commande `by []` se charge de fermer ce but et de revenir au but qui a été mis en suspens. L'hypothèse supplémentaire figure désormais dans le contexte.

Ligne 8. À cette étape, la conclusion est de la forme `n.+1 + 0 = n.+1`. Dans cette égalité, remplaçons le terme `n.+1 + 0` par `(n + 0).+1`. La réécriture via l'égalité `HypSuppl` vise cet objectif. Provisoirement, la conclusion devient `(n + 0).+1 = n.+1`. Dans le membre de droite de cette égalité, on remplace le terme `n` par `n + 0`. La réécriture via l'égalité donnée par l'hypothèse de récurrence (de droite vers la gauche) vise cet objectif. À l'issue de cette double réécriture, la conclusion du but est `(n + 0).+1 = (n + 0).+1`.

Ligne 9. La conclusion du but courant étant une égalité entre deux termes identiques, elle peut être *déchargée*. À nouveau, le très bref `by []` remplit cette fonction. Un message apparait avec un contenu explicite : *No more goals*.

Ligne 10. Il est temps d'invoquer la fin de la preuve avec la commande `Qed` (pour *Quod erat demonstrandum*). Le logiciel synthétise alors l'ensemble de la preuve en un terme qui est du type souhaité. Si on saisit la commande `Print lemme_addition_0_neutre_a_droite_bis` alors ce terme est affiché. À des ajustement près, on reconnait dans ce qui affiché la structure du terme qui a été produit dans la version initiale de l'énoncé.

Remarque 1. Avec un peu de pratique, la preuve ci-dessus aurait pu être écrite sur une ligne :

```
1 Lemma lemme_addition_0_neutre_a_droite_ter : ∀ n : nat, n + 0 = n.
2 Proof.
3   by elim => [//| n HRec]; rewrite -[in RHS]HRec addSn.
4 Qed.
```

Remarque 2. Considérons l'énoncé jumeau :

```
1 Lemma lemme_addition_0_neutre_a_gauche : ∀ n : nat, 0 + n = n.
2 Proof.
3   by [].
```

```
4  Qed.
```

Nos deux lemmes sont de faux jumeaux. Celui que l'on vient d'écrire est trivial. En effet, par construction de la fonction d'addition lorsque le premier argument est nul, la valeur renvoyée est égale au second argument. On dit que le terme `n + 0` est convertible en `n`. Dans la logique de *Coq* deux termes convertibles peuvent se substituer l'un à l'autre. Cet énoncé se ramène donc à à la proposition triviale : `∀ n : nat, n = n`.

Nous avons rencontré une situation similaire dans la justification de `HypSuppl : n.+1 + 0 = (n + 0).+1`. Par construction de la fonction d'addition, lorsque le premier argument est de la forme `n+.1` et que le deuxième argument est `n2` alors la valeur renvoyée est `(n + n2).+1`. Ainsi le terme `n.+1 + 0` est convertible en `(n + 0).+1`. Cela éclaire le caractère trivial de la justification.

2.4 *Coq* et ses tactiques

On prolonge la section précédente par un examen des tactiques disponibles dans le cours de la production d'une preuve. Si leur nombre est restreint, il convient de souligner que les modalités de leurs utilisations sont variées. Ici, le choix retenu est de décrire le fonctionnement de ces tactiques dans le cadre particulier, celui de l'extension *ssreflect*, qui diffère du cadre pur *Coq*.

2.4.1 La tactique *move*

Une preuve comporte généralement des «transferts» de termes entre la conclusion du but et le contexte, dans un sens comme dans l'autre. La tactique `move` réalise ces transferts. De manière plus précise, lorsqu'on bascule un terme de la conclusion vers le contexte, on parle d'*introduction* et l'utilisation d'un terme dans le contexte qui affecte la conclusion est qualifiée d'*élimination*.

Partons d'un exemple issu de la logique propositionnelle. Rappelons que la notation flèche désigne autant une fonction qu'une implication.

```
1  Lemma lemme_transitivite_implication : ∀ P Q R: Prop, (P -> Q) -> (Q -> R) -> P -> R.
2  Proof.
3    move => P Q R H1 H2 p.
4    exact: (H2 (H1 p)).
5  Qed.
```

Pour démarrer la preuve il faut commencer par *introduire* dans le contexte du but les variables *P*, *Q*, *R*, les hypothèses `P -> Q` et `Q -> R`. La commande `move => P Q R H1 H2` réaliserait ce transfert vers le contexte. Suite à cette opération, la conculsion serait `P -> R`. Comme, il est clair, que l'on a besoin de composer les implications `H1` et `H2`, il est également nécessaire d'introduire dans le contexte un élément dont le type est `P`. Voici à quoi ressemble le but après la validation de la ligne 3 :

```
1 goal (ID 780)

  P, Q, R : Prop
  H1 : P -> Q
  H2 : Q -> R
  p : P
  ============================
  R
```

On verra un peu plus bas que la tactique *apply :* réalise l'élimination de l'implication. D'autres exemples en lien avec la logique propositionnelle articulent ces deux tactiques.

2.4.2 La tactique *exact :*

Comme on a pu le voir, lorsqu'on est en mesure d'anticiper sur un terme dont le type répond à ce qui figure dans la conclusion, on emploie la tactique `exact:` avec le terme idoine en argument. La modalité d'utilisation de cette tactique est réduite à cette unique situation.

2.4.3 La tactique *case*

Cette tactique est en charge de la déconstruction d'un terme. Par exemple, lorsqu'on dispose d'une variable booléenne, on peut souhaiter la déconstruire pour obtenir deux sous-buts : l'un dans lequel cette variable prend la valeur `true` et l'autre dans lequel elle prend la valeur `false`. Voici un exemple :

```
1  Lemma lemme_tautologie_boolenne : ∀ b1 b2: bool, (b1 || b2) && b1 == b1.
2  Proof.
3    move => b1 b2.
4    case b1.
5      by [].
6    case b2.
7      by [].
8      by [].
9  Qed.
```

À la ligne 3, on élimine la quantification universelle par la tactique `move`. La conclusion du but devient alors `(b1 || b2) && b1 == b1`. On décide de *déconstruire* la variable `b1`. Compte-tenu de la définition du type booléen, on va obtenir deux sous-buts qui correspondent chacun à l'un des constructeurs de ce type. À cette étape, l'interface de la preuve ressemble alors à ceci :

```
  2 goals (ID 788)

  b2 : bool
  ============================
  (true || b2) && true == true

goal 2 (ID 789) is:
 (false || b2) && false == false
```

Par calcul le terme `(true || b2) && true == true` est réduit à `true == true` (ici il s'agit de l'égalité booléenne). Ce sous-but est clos. Le suivant admet pour conclusion `(false || b2) && false == false`. Or, tant que la valeur de `b2` est indéterminée, ce terme n'est pas réductible, d'où la nécessité de déconstruire ce terme. Ensuite, les deux sous-buts produits ne comportent que des constantes, ils sont donc réductibles par calcul.

2.4.4 La tactique *apply :*

Reprenons l'énoncé sur la transitivité de l'implication propositionnelle. Dans la preuve précédente, après l'introduction des variables propositionnelles, des implications `H1 : P -> Q`, `H2 : Q -> R` et de la variable `p : P`, on a procédé par présentation d'un terme dont le type répondait exactement à la conclusion. On aurait pu faire autrement en *appliquant* les implications. On dit également qu'on élimine ces *implications*.

```
1  Lemma lemme_transitivite_implication_bis : ∀ P Q R: Prop, (P -> Q) -> (Q -> R) -> P -> R.
2  Proof.
3    move => P Q R H1 H2 p.
4    apply: H2.
5    apply: H1.
```

```
6     by [].
7   Qed.
```

À la ligne 3, l'invocation de l'implication `H2 : Q -> R` transforme la conclusion du but en `Q`. À la ligne 4, l'invocation de l'implication `H1 : P -> Q` transforme la conclusion du but en `P`. Or, dans le contexte de la preuve, il existe un terme `p:P`. Cela est détecté par la commande finale `by []`.

Il existe une autre situation dans laquelle le recours à la tactique `apply:` s'impose. Lorsqu'il s'agit d'avancer dans une étape de la preuve en invoquant un énoncé disponible dans le contexte courant. Voici un exemple qui fait appel à un énoncé disponible dans la bibliotheque initialement chargée :

```
1   Check leq_mul.
```

```
: ∀ [m1 m2 n1 n2 : nat], m1 <= n1 -> m2 <= n2 ->
    m1 * m2 <= n1 * n2
```

```
1   Lemma lemme_renversant_01 : ∀ p1 p2: nat, p1 <= 3 -> 4 <= p2 -> p1 * 4 <= 3 * p2.
2   Proof.
3     move => p1 p2 H1 H2.
4     apply: leq_mul.
5     by [].
6     by [].
7   Qed.
```

Dans cet exemple, la tactique `apply: leq_mul` peut être invoquée car le terme disponible dans la conclusion `4 <= p2 -> p1 * 4 <= 3 * p2` est une spécialisation du terme `m1 * m2 <= n1 * n2`. *Coq* ouvre deux sous buts dont les conclusions correspondent aux spécialisations de `m1 <= n1` et `m2 <= n2`.

Dans cet exemple, la preuve aurait pu être écrite de manière plus concise en :

```
1   Lemma lemme_renversant_01_bis : ∀ p1 p2: nat, p1 <= 3 -> 4 <= p2 -> p1 * 4 <= 3 * p2.
2   Proof.
3     move => p1 p2.
4     apply: leq_mul.
5   Qed.
```

Coq est assez habile pour effectuer la spécialisation de `m1 <= n1 -> m2 <= n2 -> m1 * m2 <= n1 * n2`. Toutefios, l'unification des types des différents termes fait que si l'énoncé est légèrement modifié alors preuve initiale ne s'applique plus en l'état :

```
1   Lemma lemme_renversant_01_hic : ∀ p1 p2: nat, p1 <= 3 -> 4 <= p2 -> 4 * p1 <= 3 * p2.
2   Proof.
3     move => p1 p2 H1 H2.
4     apply: leq_mul.
```

L'interface de la preuve donne ceci :

```
  2 goals (ID 973)

  p1, p2 : nat
  H1 : p1 <= 3
  H2 : 3 < p2
  ============================
  3 < 3

goal 2 (ID 974) is:
```

```
p1 <= p2
```

Il ne sera pas évident de prouver que `3 < 3`... La rectification passe bien sûr par un recours *explicite* à réécriture liée à la propriété de commutativité de la multiplication des entiers :

```
Lemma lemme_renversant_01_hic_corrige : ∀ p1 p2: nat, p1 <= 3 -> 4 <= p2 -> 4 * p1 <= 3 * p2.
Proof.
  move => p1 p2 H1 H2.
  rewrite mulnC. (* transforme "4 * p1" en "p1 * 4" *)
  by apply: leq_mul.
Qed.
```

2.4.5 La tactique *elim*

La tactique `elim` met en place un raisonnment par induction. Il faut pour cela que le terme de tête dans la conclusion soit une variable mais cette variable n'est pas nécessairement du type entier. Pa exemple, il ext possible de recourir à cette tactique sur une variable de type booléen. Dans ce dernier cas, cela revient à une déconstruction, telle que celle opérée par la tactique *case*. Reprenons

```
Lemma lemme_tautologie_boolenne_bis : ∀ b1 b2: bool, (b1 || b2) && b1 == b1.
Proof.
  move => b1 b2.
  elim: b1.
  by [].
  by elim b2.
Qed.
```

La pseudo tactique `by` peut être placée en position initiale lorsque la tactique qui vient après produit des buts triviaux.

2.4.6 La tactique *rewrite*

Comme son nom l'indique, cette tactique effectue une réécriture. Lorsque l'on dispose d'un terme qui est une égalité, `Egalite: x = y`, on peut substituer, dans la conclusion, la première occurence de `x` par `y` à l'aide de la commande `rewrite Egalite`. On peut substituer, la première occurence de `y` par `x` à l'aide de la commande `rewrite -Egalite`. Si on souhaite que toutes les occurences de `x` soient remplacées par `y` alors la syntaxe est rewrite{!Egalite}. Le mécanisme de réécriture est assez souple pour être appliqué dans un motif particulier dans la conclusion.

Par exeemple dans le code rencontré auparavant, on avait :

```
Lemma etape_heredite : ∀ n : nat, n + 0 = n -> n.+1 + 0 = n.+1.
Proof.
  move => n H.
  have Hyp : n.+1 + 0 = (n + 0).+1 => //.
  rewrite Hyp -[in RHS]H //.
Qed.
```

L'hypothèse de récurrence est l'égalité `H: n + 0 = n`. La syntaxe montre que la réécriture par cette hypothèse est effectuée dans le membre de droite. En réalité, la notion de réécriture couvre un champ qui dépasse celui des égalités mais cela est hors de propos.

3 Points qui auraient dûs être abordés...

Les notions de conjonction, de disjonction et de négation propositionnelles.

La notion de quantification existentielle.

La différence entre logique classique et logique intuistionniste.

La soustraction sur les entiers naturels et les relations d'ordre boolénnes qui en découlent.

Et surtout, donner de vraies preuves, comme celle-ci (emprunté à A. Mahboubi et E. Tassi) :

```
Inductive ex2 A P Q : Prop := ex_intro2 x of P x & Q x.
Notation "exists2 x , p & q" := (ex2 (fun x => p) (fun x => q)).

Notation "n `!" := (factorial n).

(* Lemmes utilises ....

Lemma fact_gt0 n : 0 < n`!.
Lemma dvdn_fact m n : 0 < m <= n -> m %| n`!.
Lemma pdivP n : 1 < n -> exists2 p, prime p & p %| n,
Lemma dvdn_addr m d n : d %| m -> (d %| m + n) = (d %| n).
Lemma gtnNdvd n d : 0 < n -> n < d -> (d %| n) = false.

*)

Lemma prime_above m : exists2 p, m < p & prime p.
Proof.
have /pdivP[p pr_p p_dv_m1]: 1 < m`! + 1.
  by rewrite addn1 ltnS fact_gt0.
exists p => //; rewrite ltnNge; apply: contraL p_dv_m1 => p_le_m.
by rewrite dvdn_addr ?dvdn_fact ?prime_gt0 // gtnNdvd ?prime_gt1.
Qed.
```

4 Exercices

Comment résister à la tentation de donner des exercices ?

4.1 Autour de la logique booléenne

Prouver les énoncés suivants :

```
Lemma distributive_boolenne_1 : ∀ a b c : bool, a && (b || c) ==> (a && b) || (a && c).
Proof.
  Admitted.

Lemma distributive_boolenne_2 : ∀ a b c : bool, a || (b && c) ==> (a || b) && (a || c).
Proof.
  Admitted.
```

4.2 Autour de la logique propositionnelle minimale

Prouver les énoncés suivants :

```
Lemma imp_dist : ∀ P Q R : Prop,  (P -> Q -> R) -> (P -> Q) -> P -> R.
Proof.
Admitted.

Lemma imp_trans : ∀ P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R.
Proof.
Admitted.

Lemma imp_perm : ∀ P Q R : Prop, (P -> Q -> R) -> Q -> P -> R.
Proof.
Admitted.

Lemma ignore_Q : ∀ P Q R : Prop,  (P -> R) -> P -> Q -> R.
Proof.
Admitted.

Lemma delta_imp : ∀ P Q : Prop, (P -> P -> Q) -> P -> Q.
Proof.
Admitted.

Lemma delta_impR : ∀ P Q : Prop, (P -> Q) -> P -> P -> Q.
Proof.
Admitted.

Lemma diamond : ∀ P Q R : Prop, (P -> Q) -> (P -> R) -> (Q -> R -> S) -> P -> S.
Proof.
Admitted.

Lemma weak_peirce : ∀ P Q : Prop, ((((P -> Q) -> P) -> P) -> Q) -> Q.
Proof.
Admitted.
```

4.3 Autour des entiers naturels

Prouver les énoncés suivants :

```
Lemma exo_01 : ∀ n1 n2 : nat, n1 + n2 = n2 + n1.
Proof.
Admitted.

Lemma exo_02 : ∀ n1 n2 m3: nat, n1 + n2 + n3 = n1 + (n2 + n3).
Proof.
Admitted.

Lemma exo_03 : ∀ n1 n2: nat, n1 + n2 + n3 = n1 + (n2 + n3).
Proof.
Admitted.

Lemma exo_04 : ∀ n1 : nat, n1 * 0 = 0.
Proof.
Admitted.

Lemma exo_05 : ∀ n1 n2 : nat, n1 * n2.+1 = n1 * n2 + n2.
Proof.
Admitted.

Lemma exo_06 : ∀ n1 n2 n3 : nat, n1 * (n2 + n3) = n1 * n2 + n1 * n3.
Proof.
Admitted.

Lemma exo_07 : ∀ n1 n2 : nat, n1 * n2 = n2 * n1.
```

```
Proof.
Admitted.

Lemma exo_08 : ∀ n1 n2 n3 : nat, (n1 + n2) * n3 = n1 * n3 + n2 * n3.
Proof.
Admitted.
```

4.3.1 Énoncés de la logique intuistionniste

Prouver des énoncer comme :

```
Lemma and_assoc : P /\ (Q /\ R) -> (P /\ Q) /\ R.
Proof.
Admitted.

Lemma and_imp_dist : (P -> Q) /\ (R -> S) -> P /\ R -> Q /\ S.
Proof.
Admitted.

Lemma not_contrad :  ~(P /\ ~P).
Proof.
Admitted.

Lemma or_and_not : (P \/ Q) /\ ~P -> Q.
Proof.
Admitted.

Lemma not_not_exm : ~ ~ (P \/ ~ P).
Proof.
Admitted.

Lemma de_morgan_1 : ~(P \/ Q) -> ~P /\ ~Q.
Proof.
Admitted.

Lemma de_morgan_2 : ~P /\ ~Q -> ~(P \/ Q).
Proof.
Admitted.

Lemma de_morgan_3 : ~P \/ ~Q -> ~(P /\ Q).
Proof.
Admitted.

Lemma or_to_imp : P \/ Q -> ~ P -> Q.
Proof.
Admitted.

Lemma imp_to_not_not_or : (P -> Q) -> ~ ~(~P \/ Q).
Proof.
Admitted.
```

4.3.2 Énoncés de la logique classique

```
Context (tiers_exclus : ∀ P : Prop, P \/ ~P).

Lemma de_morgan_4 : ~(P /\ Q) -> ~P \/ ~Q.
Proof.
Admitted.
```

4.3.3 Énoncés autour de la surjectivité et de la surjectivité des fonctions

```
Section CompositionSurjectiviteEtInjectivite.
Variables (f g : nat -> nat).
Hypothesis gf_surjective : ∀ (c : nat), exists a:nat, g (f a) = c.
Hypothesis gf_injective : ∀ (x y : nat),  g (f x) = g (f y) -> x = y.

Lemma g_surjective : ∀ c : nat, exists b:nat, g b = c.
Proof.
Admitted.

Lemma f_injective : ∀ x y : nat, f x = f y -> x = y.
Proof.
Admitted.

Check g_surjective.
Check f_injective.
End CompositionSurjectiviteEtInjectivite.
```