De Coq à Lambdapi

Amélie Ledein

L'objectif de ce premier travail est de se familiariser à écrire des programmes et des preuves en Lambdapi, abréger LP par la suite. Pour ce faire, nous sommes partis de programmes et preuves existants, écrits en Coq, et nous avons essayé de les traduire dans LP. Plusieurs options disponibles dans les scripts Coq ne le sont pas dans LP à l'heure actuelle, et ce document tente de les mettre en lumière et ce, le plus exhaustivement possible.
Nos codes se trouvent sur `https://github.com/amelieled/lambdapi_examples`.

Avant de continuer la lecture, veuillez noter qu'un *symbole* est un ou plusieurs caractères auxquels nous associons au minimum un type[1], tandis qu'une *notation* est un raccourci d'écriture d'un symbole. Dans LP, nous associons à une notation au moins une `rule` pour faire disparaitre la notation au sein de la preuve, mais aucun principe.

1 Écrire des programmes

LP étant un "logical framework", il faut d'abord définir les symboles et notations qui seront utilisés. Dans le fichier `Notation.lp`, nous définissons ainsi le type des propositions (`Prop`), le type des données (`Set`) et l'égalité de Leibniz (`eq`). Ces notations seront utilisées partout dans la suite de ce document, nous les rappelons donc ici :

```
// Proposition type
constant symbol Prop : TYPE       // Type of propositions
set declared "π"
injective symbol π : Prop → TYPE // Interpretation of
                                  // propositions in TYPE
// Set type
constant symbol Set : TYPE        // Type of set codes
set declared "τ"
injective symbol τ : Set → TYPE  // Interpretation of
                                  // set codes in TYPE
// Leibniz equality
constant symbol eq {a} : τ a → τ a → Prop
set infix 8 "=" ≔ eq
constant symbol eq_refl {a} (x : τ a) : π (x = x)
constant symbol eq_ind {a} (x y : τ a) :
                        π (x = y) → Πp, π (p y) → π (p x)
```

[1] Nous pouvons aussi lui associer des règles de réécriture ou des principes.

Ensuite, il faut également définir formellement la logique dans laquelle nous allons travailler. C'est ce dont nous parlerons dans la section suivante. Dans cette section, nous allons brièvement présenter quelques exemples de définitions de types, fonctions et principes, etc.

1.1 De premiers exemples simples

Dans cette sous-partie, nous considérons l'exemple des booléens disponible dans le fichier `Bool.lp`. Dans LP, nous les définissons de la manière suivante :

```
constant symbol bool : Set        // Type of booleans
definition B ≔ τ bool
constant symbol true  : B
constant symbol false : B
```

Il faut noter que la deuxième ligne est équivalente à :

```
symbol B : TYPE
rule  B ↪ τ bool
```

D'autres exemples de types inductifs non récursifs sont disponibles dans le fichier `Non-recursive_type.lp`.

Une fois cela fait, nous pouvons définir des fonctions classiques travaillant sur des booléens, à l'aide de règles de réécriture :

```
symbol orb : B → B → B // Disjunction
set infix left 6 "|" ≔ orb
rule true  | _     ↪ true
with _     | true  ↪ true
with false | $b    ↪ $b
with $b    | false ↪ $b

symbol andb : B → B → B // Conjunction
set infix left 7 "&" ≔ andb
rule true  & $b    ↪ $b
with $b    & true  ↪ $b
with false & _     ↪ false
with _     & false ↪ false
```

Vous trouverez d'autres exemples de fonctions dans le fichier `Operations.lp`.

Il faut bien noter que les structures de filtrage `match...with` présentent dans Gallina ou OCaml ne peuvent pas être traduites directement en règles de réécriture. En effet, dans LP, les fonctions sont définies par des règles de réécriture indépendantes les unes des autres, c'est-à-dire que l'ordre des règles n'est pas pris en compte, tandis que dans Coq, les fonctions sont définies par point fixe à l'aide d'un filtrage qui permet d'énumérer tous les cas possibles d'un type inductif et où l'ordre des cas est important.

Ainsi, dans LP, nous devons écrire :

```
symbol isblue : C → B
rule isblue black           ↪ false
with isblue white           ↪ false
with isblue (primary blue)  ↪ true
with isblue (primary red)   ↪ false
with isblue (primary green) ↪ false
```

et non :

```
symbol isblue : C → B
rule isblue black           ↪ false
with isblue white           ↪ false
with isblue (primary blue)  ↪ true
with isblue (primary _)     ↪ false
```

Avec la même syntaxe que précédemment, nous pouvons également définir un principe d'induction sur ces booléens (qui correspond également à un principe d'analyse par cas, ici, car les constructeurs sont tous constants) :

```
// Induction principle on B.
symbol bool_ind : Πp, π(p true) → π(p false) → Πb, π(p b)
rule bool_ind _ $t _  true  ↪ $t
with bool_ind _ _  $f false ↪ $f
```

Vous trouverez d'autres exemples simples de principe d'induction dans le fichier `Color.lp`.

Dans le même esprit, nous pouvons définir des listes de booléens et c'est ce qui est fait dans le fichier `Bool_list.lp`. Ici, nous présentons plutôt la version polymorphique des listes :

```
constant symbol list : Set → Set
set declared "L"
definition L a := τ (list a)
set declared "□"
constant symbol □ {a} : L a
constant symbol cons {a} : τ a → L a → L a
set infix right 4 "::" := cons
```

1.2 Réécriture conditionnelle

Concernant les règles de réécriture, il est parfois nécessaire de pouvoir distinguer si `$x` et `$y` sont différents ou non. Ce cas particulier de réécriture conditionnelle peut être résolu en introduisant le `if` polymorphique suivant :

```
symbol if {a} : B → a → a → a // Polymorphic if
rule if true  $t _  ↪ $t
with if false _  $f ↪ $f
```

Par exemple, nous pouvons modéliser un *bag* à l'aide d'une liste, et utiliser notre `if` polymorphique pour compter le nombre d'occurrences dans ce bag d'un élément donné :

```
definition bag := τ (list nat) // Type of bags

symbol count : N → bag → N
rule count _     □    ↪ 0
with count $x ($y::$q) ↪
        if (eqb_nat $x $y) (succ (count $x $q))
                           (count $x $q)
```

Vous trouverez d'autres exemples de fonctions sur les bags dans le fichier : `Bag_via_list.lp`

2 Écrire des preuves

En toute logique, avant de pouvoir écrire des preuves, il faut définir formellement le système logique que nous considérons. Ici, comme nous nous plaçons dans le cadre de la logique intuitionniste du 1^{er} ordre, nous devons définir les règles qui permettent de manipuler les opérateurs logiques suivants : $\wedge$, $\vee$, $\bot$, $\top$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\forall$ et $\exists$.

Pour $\wedge$, $\vee$ et $\exists$, il est aisé d'écrire les règles habituelles :

```
// Conjunction
constant symbol conj : Prop → Prop → Prop
set infix right 7 "∧" := conj
constant symbol conj_intro p q : π p → π q → π (p ∧ q)
symbol conj_elim_left  p q : π (p ∧ q) → π p
symbol conj_elim_right p q : π (p ∧ q) → π q

// Disjunction
constant symbol disj : Prop → Prop → Prop
set infix right 6 "∨" := disj
constant symbol disj_intro_left  p q : π p → π (p ∨ q)
constant symbol disj_intro_right p q : π q → π (p ∨ q)
symbol disj_elim p q r :
      π (p ∨ q) → (π p → π r) → (π q → π r) → π r

// Existential quantification
set declared "∃"
constant symbol ∃ {a} : (τ a → Prop) → Prop
constant symbol ex_intro {a} p (x:τ a) : π (p x) → π (∃ p)
symbol ex_elim {a} p q :
     π (∃ p) → (Πx:τ a, π (p x) → π q) → π q
```

Il en est de même pour la règle spécifique de la logique intuitionniste associée à $\bot$:

```
// False
set declared "⊥"
constant symbol ⊥ : Prop
symbol false_elim p : π ⊥ → π p
```

Il est également nécessaire de considérer une règle pour $\top$, notamment pour obtenir l'équivalent de la tactique Coq `discriminate` (Cf 2.3.9) :

```
// True
set declared "⊤"
symbol ⊤ : Prop
symbol I : π (⊤)
```

Cependant, pour ¬, ⇒, ⇔ et ∀, il faut noter une subtilité. Nous pourrions écrire également les règles habituelles pour ces opérateurs, mais le fait de les considérer comme des notations, allège les preuves. En effet, les définitions réalisées avec le mot-clef `symbol` doivent être considérées comme des principes ou règles qu'il est nécessaire d'appliquer avec les tactiques `apply`, `rewrite` ou `refine` sur le but afin de le transformer. Au contraire, les définitions réalisées avec le mot-clef `rule` seront appliquées directement par le système.
Nous pouvons donc définir ⇔ comme une notation, comme cela est très souvent fait :

```
// Equivalence
definition equiv (p q : Prop) := (p ⇒ q) ∧ (q ⇒ p)
set infix left 7 "⇔" := equiv
```

Nous avons fait exactement la même chose pour ∀, puisqu'il existe un symbole propre à LP auxquel nous pouvons nous ramener :

```
// Universal quantification
constant symbol ∀ {a} : (τ a → Prop) → Prop
rule π (∀ $f) ↪ Πx, π ($f x)
```

Cependant, nous aurions voulu faire de même pour ¬ et ⇒ mais pour des raisons illustrées en 2.2, nous avons dû ajouter un principe à chacun de ces symboles :

```
// Negation
set declared "¬"
constant symbol ¬ : Prop → Prop
symbol neg_elim  p : π (¬ p) → π p → π ⊥
rule π (¬ $p) ↪ π $p → π ⊥

// Intro and Elimination of imp
constant symbol imp : Prop → Prop → Prop
set infix right 5 "⇒" := imp
symbol imp_elim  p q : π (imp p q) → π p → π q
rule π ($p ⇒ $q) ↪ π $p → π $q
```

Toutes les définitions qui viennent d'être présentées sont disponibles dans le fichier `Constructive_logic.lp`. Une extension à la logique classique est disponible dans le fichier `Classical_logic.lp`.

Nous avons à présent tous les outils en main pour faire des preuves. Dans les sous-sections qui suivent, nous expliquons les points communs et différences qui ont pu être observés entre des preuves écrites en Coq et des preuves écrites en LP.

2.1 A propos du jeu de tactiques

A l'heure où j'écris ces lignes, seules les tactiques suivantes existent dans LP :

- Tactiques de management de preuves : `print`, `focus`
- Tactique d'introduction de variables ou d'hypothèses : `assume`
- Tactique de simplification : `simpl`
- Tactiques générales : `refine`, `apply`
- Tactiques autour de l'égalité : `rewrite`, `reflexivity`, `symmetry`
- Tactique appelant des prouveurs automatiques : `why3`
- Tactiques clôturant un script de preuve : `abort`, `admit`, `qed`.

Il n'y a donc pas, en particulier, les tactiques : `assert`, `cut`, `case`, `induction`, `discriminate`, `injection`, `inversion`, et bien d'autres[2].

Ayant posé notre système logique ainsi que les tactiques utilisables dans LP, nous pouvons à présent observer les différences existantes entre les preuves écrites en Coq et celles écrites dans LP. Cela constitue le sujet des sous-sections suivantes.

2.2 Concernant le style de preuve

Il faut noter qu'**il n'est actuellement pas possible de faire des "forward-style proofs" avec Lambdapi**, ce qui rend la traduction entre les preuves écrites en Coq et les preuves écrites en Lambdapi moins directe.
En effet, il n'est pas possible d'appliquer les tactiques qui le peuvent sur les hypothèses. Ainsi, toute tactique s'applique sur le but et ne peut donc modifier que celui-ci ou échouer.

Prenons un exemple issu du fichier `Logic_lemma.lp` :

```
theorem P_notP_contradiction : ΠP : Prop, π (¬(P ∧ ¬P))
```

Durant la preuve de celui-ci, nous serions bloqués si nous n'avions pas ajouté la règle suivante à ¬[3] :

```
symbol neg_elim  p : π (¬ p) → π p → π ⊥
```

En effet, nous obtenons le but suivant :

```
== Goals ================================
   P : Prop
   Hand : π (P ∧ ¬ P)
   ----------------------------------------
0. π ⊥
```

Pour terminer cette preuve, nous devons utiliser `apply neg_elim P` puis les règles `conj_elim_right` et `conj_elim_left`, comme nous ne pouvons pas éliminer ∧ de l'hypothèse `Hand`, puis faire `apply HnotP apply HP qed`.

[2] Un tableau récapitulatif se trouve en fin de document.
[3] Nous avons fait de même pour ⇒ (Cf le fichier `Classical_logic.lp`).

Un autre exemple, plus compliqué, nous montre que l'absence de pouvoir faire des preuves en avant nous oblige à modifier certaines preuves. Considérons la preuve suivante en Coq :

```
Lemma andb_prop : forall b1 b2,
  andb b1 b2 = true → b1 = true ∧ b2 = true.
Proof.
  intros b1 b2 H. (* introduit les hypotheses *)
  split. (* separe le but en deux sous-buts *)
  - destruct b1. (* raisonnement par cas *)
    + reflexivity. (* true = true *)
    + simpl in H. discriminate H. (* false <> true *)
  - destruct b2. (* raisonnement par cas *)
    + reflexivity. (* true = true *)
    + destruct b1 .  (* raisonnement par cas *)
      * simpl in H. discriminate H. (* false <> true *)
      * simpl in H. discriminate H. (* false <> true *)
Qed.
```

Il semble plus facile et naturel en LP, d'introduire une analyse par cas (ou notre principe d'induction) juste après la tactique `assume` qui introduit uniquement des variables, ce qui évite plusieurs réécritures manuelles :

```
theorem andb_prop2 b1 b2
  : π (imp (andb b1 b2 = true) ((b1 = true) ∧ (b2 = true)))
proof
  assume b1 b2
  refine bool_ind (λz, imp (andb z b2 = true)
                           ((z = true) ∧ (b2 = true))) _ _ _
  // Case b1 = true
  simpl
  assume H
  apply conj_intro
  reflexivity
  apply H
  // Case b1 = false
  simpl
  assume H
  apply false_elim (false = true ∧ b2 = true)
  apply discr_f_t apply H
qed
```

Nous pouvons utiliser la tactique `split` au même niveau que dans le script Coq, mais cela bouleverse totalement la suite de la preuve dans LP (nous rappelons la preuve Coq pour plus de lisibilité) :

```
theorem andb_prop b1 b2
  : π (imp (andb b1 b2 = true) ((b1 = true) ∧ (b2 = true)))
proof
  assume b1 b2 H1 apply conj_intro
  // 0. π (b1 = true)
  refine bool_case (λz, z = true) b1 _ _  // destruct b1
         // 0. π (b1 = true) → π (b1 = true)
         assume H2 apply H2
         // 1. π (b1 = false) → π (b1 = true)
         assume H2
         apply eq_ind true (andb b1 b2) _ (λz, b1 = z)
               // 0. π (true = andb b1 b2)
               rewrite H1 reflexivity
               // 1. π (b1 = andb b1 b2)
               rewrite H2 reflexivity
  // 1. π (b2 = true)
  refine bool_case (λz, z = true) b2 _ _ // destruct b2
        // 0. π (b2 = true) → π (b2 = true)
        assume H2 apply H2
        // 1. π (b2 = false) → π (b2 = true)
        assume H2
        apply eq_ind true (andb b1 b2) _ (λz, b2 = z)
               // 0. π (true = andb b1 b2)
               rewrite H1 reflexivity
               // 1. π (b2 = andb b1 b2)
               rewrite H2 reflexivity
qed
```

```
1  Lemma andb_prop : forall b1 b2,
2    andb b1 b2 = true → b1 = true ∧ b2 = true.
3  Proof.  intros b1 b2 H. (* introduit les hypotheses *)
4    split. (* separe le but en deux sous-buts *)
5    - destruct b1. (* raisonnement par cas *)
6      + reflexivity. (* true = true *)
7      + simpl in H. discriminate H. (* false <> true *)
8    - destruct b2. (* raisonnement par cas *)
9      + reflexivity. (* true = true *)
10     + destruct b1 .  (* raisonnement par cas *)
11       * simpl in H. discriminate H. (* false <> true *)
12       * simpl in H. discriminate H. (* false <> true *)
13 Qed.
```

Vous noterez l'utilisation de `discr_f_t` qui correspond à un cas particulier d'utilisation de la tactique Coq `discriminate` sur les booléens, dont nous parlerons dans en 2.3.9. D'autres exemples d'utilisation de `discr_f_t` sont également visibles dans le fichier `Bool_lemma.lp`.

2.3 A la recherche des liens entre les tactiques de Coq et de LP

Les 4 premières sous-sous-sections traitent d'exemples tirés de la logique, les 2 suivantes discutent de tactiques dont l'absence pose quelques problèmes, puis les 4 dernières sous-sous-sections concernent plus les tactiques liées aux types inductifs.

2.3.1 Principe d'élimination

A travers le 1^{er} exemple de la sous-section précédente, nous avons déjà vu que l'élimination du $\wedge$ se fait à l'aide des règles `conj_elim_right` et `conj_elim_left` (Cf également le `theorem impl_and` dans le fichier `Logic_lemma.lp`).
Il en est de même pour $\vee$ (`apply disj_elim`) et $\exists$ (`apply ex_elim`).

```
Definition is_even n := exists p, n = 2 * p.

Lemma is_even_plus : forall n p,
   is_even n → is_even p → is_even (n + p).
Proof.
intros n p Hn Hp.
unfold is_even in Hn.
destruct Hn as [k Hk].
destruct Hp as [t Ht].
unfold is_even.
exists (k + t).
rewrite  mult_plus_distr_l.
rewrite Hk.  rewrite Ht. reflexivity.
Qed.
```

```
definition is_even n := ∃(λp, n = (2×p))

theorem is_even_plus :
  Πn p, π (is_even n) → π (is_even p) → π (is_even (n+p))
proof
  assume n p Hn Hp
  apply ex_elim (λp, n = ((succ (succ 0))×p))
  apply Hn assume x Hx
  apply ex_elim (λp0, p = ((succ (succ 0))×p0))
  apply Hp assume y Hy
  apply ex_intro (λp0, (n+p) = ((succ (succ 0))×p0)) (x+y)
  rewrite mult_plus_distr_l
  rewrite  Hx rewrite Hy reflexivity
qed
```

De manière générale, les tactiques Coq qui permettent d'éliminer un symbole sont bien sûr mimées par le principe d'élimination associé à ce symbole, dans LP, mais la preuve peut être légèrement différente.

2.3.2 Principe d'introduction

Comme l'introduction du $\wedge$ correspond à la tactique Coq `split`, la traduction de la preuve Coq en LP est immédiate, comme le montre le théorème suivant :

```
Lemma conj_impl :
forall P Q R : Prop, (P ∧ Q → R) → (P → Q → R).
Proof.
intros P Q R Hyp1 H1 H2.
apply Hyp1.
split ; assumption.
Qed.
```

```
theorem conj_impl :
ΠP Q R : Prop, π (((P ∧ Q) ⇒ R) ⇒ (P ⇒ (Q ⇒ R)))
proof
  assume P Q R Hand HP HQ
  // Goal π ((P ∧ Q) ⇒ R)
  apply Hand
  // Goal  π (P ∧ Q)
  apply conj_intro P Q
    apply HP
    apply HQ
qed
```

Il en est de même pour les tactiques Coq `left` et `right` qui sont traduites respectivement par `apply disj_intro_left` et `apply disj_intro_right` (Cf exemple tiré du fichier `Color.lp` de la section 2.4), ainsi que pour la tactique `exists`, comme nous l'avons vu dans la sous-sous-section précédente.

De manière générale, les tactiques Coq qui permettent d'introduire un symbole sont bien sûr mimées par le principe d'introduction associé à ce symbole, dans LP.

2.3.3 La tactique `contradiction`

Il n'est pas toujours facile de donner une traduction pour la tactique `contradiction`. Lorsque nous avons P et $\neg P$ dans nos hypothèses, l'application du modus ponens nous permet de conclure.
Lorsque nous avons $\bot$ en hypothèse, `apply false_elim` nous permet de conclure.

Mais dans certains cas, la tactique `refine` cache complètement l'utilisation potentielle de la tactique `contradiction` :

```
1 Lemma even_dec : forall n, even n ∨ ~ even n.
2 Proof.
3 intro n.
4 assert(even n ∨ even (succ n)) by (apply even_n_or_succn).
5 destruct H as [| HypN].
6   * left. assumption.
7   * right. intro H. apply even_n_not_succn in H.
8     contradiction.
9 Defined.
```

```
theorem even_dec : Πn, π (even n ∨ ¬(even n))
proof
  assume n
  apply disj_elim (even n) (even (succ n))
  apply even_n_disj_succn n
     // 0. π (even n) → π (even n ∨ ¬ (even (succ n)))
     assume Hn apply disj_intro_left apply Hn
     //1. π (even (succ n)) → π (even n ∨ ¬ (even (succ n)))
     assume Hsuccn apply disj_intro_right
     assume Hn
     refine even_n_neg_succn n Hn Hsuccn
qed
```

2.3.4 Un dernier exemple tiré de la logique

Ce dernier exemple tiré de `Logic_lemma.lp`, montre à quel point la traduction ne peut être automatique, mais souligne également les choix subtiles qui doivent être faits entre les nouvelles tactiques à implémenter et les règles choisies :

```
1 Lemma exists_neg_forall : forall P : nat → Prop,
2 (exists n, P n) →  ~(forall n, ~(P n)).
3 Proof.
4 intros P H H1.
5 destruct H as [n0 H0].
6 specialize (H1 n0).
7 contradiction.
8 Qed.
```

```
theorem exists_neg_forall :
Π(P :N → Prop), π(∃P) → π (¬(forall (λm:N, ¬ (P m))))
proof
  assume P Hexists Hforall
  apply ex_elim {nat} P ⊥
  apply Hexists
```

```
  refine Hforall // or assume x H apply Hforall x H
qed
```

D'autres exemples de preuves autour de la logique sont disponibles dans le fichier `Logic_lemma.lp`, et montrent également que la tactique Coq `unfold` n'est pas traduite car elle est "cachée" dans les définitions que nous avons écrites dans `Constructive_logic.lp`, avec le mot-clef `rule`.

2.3.5 La tactique `rewrite <-`

De nombreuses preuves concernant les entiers naturels sont disponibles dans le fichier `Nat.lp`. Celles-ci parlent uniquement des opérations classiques que sont l'addition et la multiplication, mais soulignent qu'il est possible de choisir uniquement 2 règles de réécriture pour chacune de ces opérations, puis de démontrer les autres. De plus, ces preuves sont identiques à celles réalisées dans Coq, sauf lorsque la tactique `rewrite <-` a été utilisée, comme dans le `theorem mult_plus_distr_l`. En effet, si nous voulons utiliser une égalité pour transformer notre but, seul le sens direct peut être utilisé. J'ai donc parfois dû réécrire un théorème déjà prouvé dans l'autre sens, afin de pouvoir l'appliquer dans le sens souhaité[4]. Cependant, il existe une manière plus élégante de finir ces preuves : **l'application de `eq_ind` permet d'utiliser l'hypothèse dans l'autre sens, à tout instant.**
Cependant, il faut que `eq_ind` soit défini comme cela :

```
constant symbol eq_ind2 {a} (x y : τ a) :
                  π (x = y) → Πp, π (p x) → π (p y)
```

Vous trouverez un exemple d'utilisation en 2.3.7 (Cf fichier `Even.lp`).

D'autres preuves quasiment identiques sont disponibles dans le fichier `Polymorphic_list.lp`, sauf les 2 dernières preuves qui ont besoin d'adaption pour traduire la tactique `inversion`. Nous parlerons de cela dans la section 2.3.10.

2.3.6 La tactique `assert`

Il peut être très utile de pouvoir introduire à n'importe quel moment, une preuve dite *intermédiaire*. Dans le LP actuel, nous pouvons simplement sortir la preuve en question, comme nous l'avons fait pour le 1^{er} `assert` de la preuve suivante :

```
1 Lemma two_steps_induction : forall P : nat → Prop, P 0 → P 1 →
2   (forall n, P n → P (succ (succ n))) →
3   forall n, P n.
4 Proof.
5   intros P H0 H1 IHn.
6   assert (forall n, P n ∧ P (succ n)) as Hn.
7    - induction n.
8       + split;assumption.
```

[4] Ces ajouts ont pour noms `*_trash`.

```
 9        + decompose [with] IHn0. apply IHn in H.
10          split;assumption.
11    - induction n.
12        + assumption.
13        + assert(P n ∧ P (succ n)) by (apply Hn).
14          decompose [with] H. assumption.
15 Defined.
```

En dehors de la preuve de `two_steps_induction`, nous avons donc une preuve de :

```
theorem assert_for_two_steps_induction :
Πp, π (p 0) → π (p 1) →
(Πn, π (p n) → π (p (succ (succ n)))) →
(Πn, π (p n ∧ p (succ n)))}.
```

Cependant, cette astuce n'est pas suffisante pour le 2^e `assert`. Ici, nous devons "construire" l'énoncé de `assert` avant de pouvoir utiliser le théorème précédent :

```
theorem two_steps_induction : Π(p : N → Prop),
π(p 0) → π(p 1) → π (forall (λn, p n ⇒ p (succ (succ n)))) →
π (forall (λn, p n))
proof
  assume p Hp0 Hp1 IHn
  refine nat_ind _ _ _
  // 0. π (p zero)
  apply Hp0
  //1. Π(x:τ nat), π (p x) → π (p (succ x))
  //assert(P n /\ P (succ n)) by (apply Hn).
  //    decompose [with] H. assumption
  assume x Hx
  apply conj_elim_right (p x)
  refine assert_for_two_steps_induction p Hp0 Hp1 IHn x
qed
```

Cette preuve se trouve dans le fichier `Even.lp`.

A priori, rien n'assure qu'il soit toujours possible de "construire" l'énoncé que nous voulons démontrer, mais l'utilisation du modus ponens peut certainement remplacer la tactique `assert` pour l'instant.

2.3.7 La tactique `case / case_eq`

Pour réaliser des analyses par cas, nous devons définir explicitement ce principe dans LP, comme c'est également le cas pour les principes d'induction. Nous avions donné un exemple page 3 pour les booléens, qui est à la fois un principe d'analyse par cas et un principe d'induction. Lorsque nous voulons traduire les tactiques `case`, `case_eq` (ou encore `destruct`), nous devons donc appliquer un principe déjà défini.

Cependant, il est parfois utile de garder l'égalité qui indique dans quel cas de l'analyse par cas nous sommes. Cela est par exemple fait en Coq avec la tactique `destruct...eqn:H`. Pour mimer ce procédé, nous devons plutôt écrire le principe suivant :

```
symbol bool_case :
Πp b, π((b = true) ⇒ (p b)) → π((b = false) ⇒ (p b)) → π(p b)
```

Voici un exemple de preuve tiré du fichier `Even.lp` qui illustre ce cas de traduction :

```
1 Lemma even_dec_bis : forall n, even n ∨ ~ even n.
2 Proof.
3 intro n. destruct (evenb n) eqn:H.
4  * left. apply evenb_correct;assumption.
5  * right. unfold not;intro.
6    apply evenb_complete in H0.
7    rewrite H in H0. inversion H0.
8 Defined.
```

```
theorem even_dec_bis : Πn, π (even n ∨ ¬(even n))
proof
  assume n
  refine bool_case (λb, (even n ∨ ¬(even n))) (evenb n) _ _
  // 0. π (evenb n = true) → π (even n ∨ ¬ (even n))
  assume Hevenbtrue apply disj_intro_left
  refine evenb_correct n Hevenbtrue
  // 1. π (evenb n = false) → π (even n ∨ ¬ (even n))
  assume Hevenbfalse apply disj_intro_right
  assume Hevenn apply discr_f_t
  refine eq_ind2 (evenb n) false Hevenbfalse (λz, z = true) _
  refine evenb_complete n Hevenn
qed
```

Il faut noter que ici, à cause du fait que nous n'avons pas la tactique `rewrite <-`, nous avons utilisé `eq_ind2`.

2.3.8 La tactique `induction`

Nous l'avons vu à plusieurs reprises dans divers exemples : pour faire une preuve par induction, il faut écrire explicitement le principe d'induction, puis nous pouvons l'utiliser notamment avec la tactique `refine`. Voici un exemple de principe d'induction pour un prédicat inductif :

```
1 Inductive even : nat → Prop :=
2  | even_0 : even 0
3  | even_SS : forall n, even n → even (succ (succ n)).
```

```
symbol even : N → Prop
constant symbol even_0_coq : π (even 0)
```

```
constant symbol even_SS_coq n : π (even n) → π (even (succ (succ n)))

symbol even_ind : Πp, π (p 0) → (Πn, π (even n) → π (p n) → π
(p (succ (succ n)))) → Πn, π (even n) → π (p n)
```

Il faut tout de même également noter que la tactique Coq `induction` réalise également des `intros` (voire des simplifications), ce qui explique que dans certaines preuves faites dans LP, nous avons dû ajouter des `assume` (voire des `simpl`) par rapport aux scripts Coq initiaux.

2.3.9 La tactique `discriminate`

Nous avons vu à plusieurs reprises que la tactique Coq `discriminate` n'existe pas dans LP. Dans le cas particulier des booléens, nous avons réalisé la preuve de l'utilisation de cette tactique :

```
symbol my_P : B → Prop
rule my_P true  ↪ ⊤
with my_P false ↪ ⊥

theorem discr_f_t : π (false = true) → π ⊥
proof
  assume H
  apply eq_ind false true H my_P
  apply I
qed
```

en rappelant la définition suivante :

```
constant symbol eq_ind {a} (x y : τ a) :
                        π (x = y) → Πp, π (p y) → π (p x)
```

Des tentatives de concision et de généralisation sont également disponibles dans le fichier `Discriminate.lp`, mais nous devons avancer le travail concernant la construction de type inductif afin de définir le prédicat P dans toute sa généralité.
Tout d'abord, nous avons défini des symboles permettant de récupérer la partie de gauche ou la partie de droite d'une hypothèse de la forme $\pi(x = y)$:

```
symbol eq_left {a} {x:τ a} {y:τ a} (H:π(x=y)) : τ a
rule eq_left {_} {$x} {_} _ ↪ $x

symbol eq_right {a} {x:τ a} {y:τ a} (H:π(x=y)) : τ a
rule eq_right {_} {_} {$y} _ ↪ $y
```

Nous pouvons donc écrire :

```
definition discr_1 {a} {x:τ a} {y:τ a} (H:π(x=y)) :=
  eq_ind (eq_left H) (eq_right H) H
```

Il nous reste juste à écrire notre prédicat général qui dépend du principe d'induction sur `Set` :

```
definition is_0 := nat_rec ⊤ (λ_ _,⊥)
```

Dans le cas des entiers naturels, nous obtenons la preuve suivante :

```
theorem succ_is_not_zero n : π (succ n ≠ 0)
proof
  assume n h
  refine eq_ind h is_0 top
qed
```

2.3.10 La tactique `inversion`

Comme pour d'autres tactiques précédemment discutées, pour traduire ce que fait la tactique Coq `inversion`, il est nécessaire de définir des principes d'inversion. Prenons l'exemple suivant (disponible dans le fichier `Polymorphic_list.lp`) :

```
1 Variable A : Type.
2 Inductive mem : A → list A → Prop :=
3   mem_head : forall x l, mem x (cons x l)
4 | mem_tail : forall x y l, mem x l → mem x (cons y l).
```

```
symbol mem {A} : τ A → L A → Prop
constant symbol mem_head {A} (x  :τ A) (l:L A) :
                                        π (mem x (x::l))
constant symbol mem_tail {A} (x y:τ A) (l:L A) :
                              π ((mem x l) ⇒ (mem x (y::l)))
```

Voici les principes d'inversion que nous avons écrit :

```
symbol notMemnil {A} : Π(x:τ A), π (mem x □) → π (⊥)
symbol mem_inv {A} (x y : τ A) (l:L A) :
      π ((mem x (y::l) ⇒ (x = y ∨ mem x l)))
```

Ceux-ci permettent que traduire assez facilement la preuve suivante :

```
1 Lemma app_left : forall l1 l2 x,
2 mem x l1 → mem x (app l1 l2).
3 Proof.
4 induction l1;intros l2 x Hyp.
5  * inversion Hyp.
6  * inversion Hyp.
7      - subst. constructor.
8      - simpl. constructor. apply IHl1. assumption.
9 Defined.
```

```
theorem app_left : ΠA (l1 l2: L A) (x:τ A),
π (mem x l1) → π (mem x (l1·l2))
proof
 assume A
  refine list_ind _ _ _
```

```
  // 0. Π(x:τ (list A)) (x0:τ A), π (mem x0 □) → π (mem x0 (□ ·
x))
  assume x x0 Hmemnil
  apply false_elim apply notMemnil x0 apply Hmemnil
  // 1. Π(x:τ A) (l:τ (list A)), (Π(x:τ (list A)) (x0:τ A), π
(mem x0 l) → π (mem x0 (l · x)))
  //            → Π(x0:τ (list A)) (x1:τ A), π (mem x1 (x :: l)) →
π (mem x1 ((x :: l) · x0))
  simpl assume x l1 IHl1 l2 x2 Hl1
  apply disj_elim (x2=x) (mem x2 l1)
        // 0. π ((x2 = x) ∨ mem x2 (l1 · l2))
        apply mem_inv x2 x l1 Hl1
        // 1. π (x2 = x) → π (mem x2 (x :: (l1 · l2)))
        assume Heqx2x rewrite Heqx2x apply mem_head x (l1·l2)
        // 2. π (mem x2 l1) → π (mem x2 (x :: (l1 · l2)))
        assume Hl1
        simpl //apply disj_intro_right apply Hmeml2
        refine mem_tail x2 x (l1·l2) _
        apply IHl1 l2 x2 Hl2
qed
```

Un autre exemple est également disponible dans le fichier `Polymorphic_list.lp`.

Il serait préférable de pouvoir démontrer ces principes, mais je n'y suis pas encore arrivée.

Nous pouvons considérer un autre exemple du fichier `Even.lp`. Nous avons écrit les principes d'inversion suivants :

```
symbol even_inv : Πn, π (even (succ (succ n))) → π (even n)
symbol notEven1 : π (even 1) → π (⊥)
```

Voici un exemple de démonstration :

```
1 Theorem even5_nonsense :
2   even 5 → 2 + 2 = 9.
3 Proof.
4   intro H.
5   inversion H.
6   inversion H1.
7   inversion H3.
8 Qed.
```

```
theorem even5_nonsense : π (even 5) → π ((2+2) = 9)
proof
  assume H5 apply false_elim
  apply notEven1
  apply even_inv (succ zero)
  apply even_inv (succ (succ (succ zero))) apply H5
qed
```

Enfin, il faut noter que parfois, le principe d'induction sur `even` permet de se passer des principes d'inversion :

```
Theorem ev_minus2 : forall n,
  even n → even (pred (pred n)).
Proof.
  intros n H.
  inversion H  as [| p Hp].
   - (* H = even_0 *) simpl. apply even_0.
   - (* H = even_SS p Hp *) simpl. exact Hp (* ou assumption*)
     .
Qed.
```

```
symbol pred : N → N
rule pred     0     ↪ 0
with pred (succ $n) ↪ $n

theorem ev_minus2 : Πn, π (imp (even n) (even (pred (pred n))))
proof
  //assume n Heven
  refine even_ind _ _ _
  //0. π (even (pred (pred zero)))
  simpl apply even_0
  //1. Π(n:τ nat), π (even n) → π (even (pred (pred n))) → π
(even (pred (pred (succ (succ n)))))
  assume n Heven Hpredpred
  simpl apply Heven
qed
```

2.4 Concernant la structure du script

Dans cette dernière sous-section, nous considérons un exemple de preuve un peu plus long. Voici le script obtenu avec Coq :

```
Theorem my_color_test :
forall c, isred c = true ∨ isgreen c =true ∨ isblue c = true
↔ monochrome c = false.
Proof.
split;intro myHyp.
 * induction c.
     + destruct myHyp as [Hred|Hgreen_blue].
       { discriminate Hred. }
       { destruct Hgreen_blue as [Hgreen|Hblue].
           - discriminate Hgreen.
           - discriminate Hblue.
       }
     + destruct myHyp as [Hred|Hgreen_blue].
       { discriminate Hred. }
       { destruct Hgreen_blue as [Hgreen|Hblue].
```

```
16              - discriminate Hgreen.
17              - discriminate Hblue.
18          }
19      + induction p;reflexivity.
20  * induction c.
21      + discriminate myHyp.
22      + discriminate myHyp.
23      + induction p.
24          - left. reflexivity.
25          - right. left.  reflexivity.
26          - right. right. reflexivity.
27 Qed.
```

Le script que nous obtenons dans LP est quasiment le même, sauf que, comme nous ne pouvons pas traduire les `destruct` directement, les preuves sont légèrement plus longues. Mais surtout, l'absence de *bullets* (+, -, *) et d'accolades font perdre en lisibilité. C'est pourquoi, j'ai "découpé" la preuve Coq en plusieurs théorèmes, afin de ne pas perdre le fil. Il est assez simple de retrouver les correspondances entre ces 2 preuves.
Le script LP suivant correspond aux lignes 23 à 26 :

```
theorem right_to_left_rgb : Πn,
π (monochrome (primary n) = false ⇒ isred (primary n) = true ∨
isgreen (primary n) = true ∨ isblue (primary n) = true)
proof
  refine rgb_ind (λz, monochrome (primary z) = false ⇒
                                    isred (primary z) = true ∨
isgreen (primary z) = true ∨ isblue (primary z) = true) _ _ _
  // Goal red
  simpl assume Hyp
  apply disj_intro_left reflexivity
  // Goal green
  simpl assume Hyp
  apply disj_intro_right apply disj_intro_left reflexivity
  // Goal blue
  simpl assume Hyp
  apply disj_intro_right apply disj_intro_right reflexivity
qed
```

Le script LP suivant reprend les lignes 20 à 26 :

```
theorem right_to_left_color :
Πc, π ((monochrome c) = false ⇒ (isred c)=true ∨ (isgreen c)=true ∨
(isblue c)=true)
proof
  assume c
  apply color_ind (λz, (monochrome z) = false ⇒ (isred z)=true ∨
(isgreen z)=true ∨ (isblue z)=true) _ _ _ c
  // Goal black
  simpl assume Htruefalse
  apply false_elim apply discr_f_t symmetry apply Htruefalse
```

```
  // Goal white
  simpl assume Htruefalse
  apply false_elim apply discr_f_t symmetry apply Htruefalse
  // Goal primary
  assume rgb
  apply right_to_left_rgb rgb
qed
```

Enfin, les lignes suivantes correspondent simplement à la ligne 19 :

```
theorem left_to_right_rgb : Π(n:RGB),
π ((isred (primary n))=true ∨ (isgreen (primary n))=true ∨ (isblue (primary n))=true
       ⇒ (monochrome (primary n))=false)
proof
  assume n
  apply rgb_ind (λz,(isred (primary z))=true ∨ (isgreen (primary z))=true ∨
(isblue (primary z))=true
       ⇒ (monochrome (primary z))=false) _ _ _ n
  simpl assume Hyp reflexivity
  simpl assume Hyp reflexivity
  simpl assume Hyp reflexivity
qed
```

La preuve complète est disponible dans le fichier `Color.lp`.

Cet exemple montre que l'ajout de bullets peut être intéressant. Simplement l'ajout des accolades peut suffire dans un premier temps.
De plus, les commandes `Search` et `Print` de Coq n'ont pas d'équivalent dans LP, contrairement à `Eval...compute` et `Check` (Cf tableau 2).
Enfin, l'ajout de tacticielles pourrait également permettre la concision des preuves, notamment la tacticielle "`;`" (Cf tableau 1), mais cela semble moins prioritaire .

Tacticielles Coq	**Tacticielles LP**
`try` *tactique*	-
`T; [T1 | T2 | ... | Tn]`	-
`repeat` *tactique*	-
`do` *entier naturel*	-

Figure 1: Isomorphisme des tacticielles Coq et LP

En conclusion, la majorité des tactiques Coq existent dans LP, ou du moins peuvent être adaptées dans LP (Cf tableau 4).

Il faut noter que le fait que nous ne pouvions faire que des preuves en arrière n'empêche a priori pas de réaliser une preuve : elles seront juste différentes de ce que nous écrirons en Coq.

Ensuite, actuellement aucune tacticielle n'existe dans LP, comme le montre le tableau 1. Les 3 dernières semblent vraiment utiles pour gagner en concision.

De plus, l'ajout d'indentation et de bullets semble important pour gagner en lisibilité du script. Seul l'ajout des accolades peut suffire pour l'instant.

Enfin, afin d'alléger l'écriture de certaines tactiques, il serait utile d'ajouter un "miniLTac", ou plutôt la possibilité de définir `Tactic Notation`.

Commandes Coq	**Commandes LP**
Requêtes	
`Search`	-
`Check`	`type`
`Print`	-
`About`	-
`Locate`	-
`Eval...in / Compute`	`compute`
`Print Assumptions`	-
`Hint`	-
`Focus`	`focus`
`Show`	-
`Section / Module`	-
`Require`	`require [open]`
`Fail`	-
`Scheme`	
`Extraction`	-
Définitions	
`Lemma / Theorem / Corollary / Proposition / Remark / Fact`	`theorem`
`Goal`	-
`Axiom(s)`	-
`Arguments`	-
`Variable(s) / Hypothesis / Hypotheses`	-
`Parameter(s)`	-
`Definition / Example`	
`Fixpoint`	
`Function`	-
`Inductive`	-
`CoInductive`	-

Figure 2: Isomorphisme des commandes Coq et LP

Commandes Coq	**Commandes LP**
`Record`	Cas particulier de `Inductive`
`Notation "A ∨ B" := (or A B)` `(at level 85, right associativity).`	`set infix left 6 "|" := orb`
`Tactic Notation`	-
`Ltac`	-

Figure 3: Isomorphisme des commandes Coq et LP (suite)

* : Ces tactiques peuvent être appliquées sur une hypothèse.
** : Il faut définir les principes à la main dans LP.

Tactiques Coq	Tactiques LP équivalentes	Adaption
Tactiques usuelles		
`intro` ou `intros`	`assume`	
`simpl`*	`simpl`	
`exact`	`refine`	
`apply...[with...]`*	`apply`	
`assumption`	-	`apply / refine`
`case / case_eq`	-	**
`destruct...as...`	-	principe d'élimination (`apply disj_elim`, etc.)
`destruct...eqn:...`	-	Cf 2.3.7
`admit`	-	X
Tactiques autour de l'égalité		
`reflexivity`	`reflexivity`	
`rewrite`*	`rewrite`	
`symmetry`*	`symmetry`	
`transitivity`	-	**
`unfold`*	-	X
`subst`	-	plusieurs `rewrite`
`replace`	-	`modus ponens` ?
`change`	-	X
`refine`	`refine` ?	
Tactiques autour des types inductifs		
`constructor`	-	`apply _`
`(double) induction...as...using...`	-	**
`elim`	-	**
`discriminate`	-	**
`injection`	-	**
`inversion`	-	**
Tactiques pour la logique		
`left`	-	`apply disj_intro_left`
`right`	-	`apply disj_intro_right`
`split`	-	`apply conj_intro _ _`
`exists`	-	`apply ex_intro`
`decompose [or] _` `decompose [and] _`	-	plusieurs `left/right` ou `split`
`exfalso`	`apply false_elim`	
`contradiction`	Cf 2.3.3	
`absurd`	idem `contradiction` ?	

Figure 4: Isomorphisme des tactiques Coq et LP

Tactiques Coq	Tactiques LP équivalentes	Adaption
Tactiques plus avancées		
`clear H`	-	X
`assert (H: e) (or assert (e) as H) / cut / (cutrewrite/enough)`	-	`modus ponens` ?
`pose proof`	-	`modus ponens` ?
`remember`	-	`modus ponens` ?
`rename...into...`	-	`modus ponens` ?
`pattern`	-	X
`specialize`	-	`refine/rewrite` ?
`generalize` `generalize dependent x)`	-	X
Tactiques automatiques		
`auto / trivial`	-	`why3`
`tauto`	-	`why3`
`omega` / `lia` (sur les entiers) `fourier` (sur les réels)	-	`why3` ?
`congruence`	-	`why3`
`intuition`	-	`why3`
`ring / field / linear`	-	`why3` ?
`smt`	`why3` ?	

Figure 5: Isomorphisme des tactiques Coq et LP (suite)