Chapter 1

theme07inductivestudent

```
Require Import Arith.
Require Import List.
```

Dans ce thème, nous allons approfondir la notion de type inductif et les techniques de démonstrations associées.

Le mot-clé important dans Coq pour ce thème est `Inductive`.

1.1 Types énumérés

Les cas les plus simples de types inductifs sont les types ... non-récursifs. C'est le cas des types énumérés comme notamment *bool*.

```
Print bool.
```

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

Prenons un autre exemple : les couleurs de jeu de carte.

```
Inductive Couleur : Set :=
  | pique : Couleur
  | coeur : Couleur
  | carreau : Couleur
  | trefle : Couleur.
```

Ici nous définissons quatre constructeurs différents pour le type *Couleur*.
Il est utile d'adopter un mode de représentation des types inductifs sous forme de règles d'inférences décrivant les constructions possibles pour le type spécifié.

La définition ci-dessus peut se “lire” ainsi :

$$\frac{}{pique : Couleur} \quad \frac{}{coeur : Couleur} \quad \frac{}{carreau : Couleur} \quad \frac{}{trefle : Couleur}$$

Pour chaque type inductif défini en Coq, l'environnement génère notamment :

- un principe de *pattern-matching* permettant de décomposer les éléments du type,
- un principe de récursion structurelle permettant d'effectuer des calculs,
- un principe d'induction permettant de raisonner sur ces mêmes calculs.

Les principes de récursion sont exploités notamment par la commande `Fixpoint`, pour définir des fonctions récursives sur un argument du type inductif considéré. Ici, comme notre type n'est pas véritablement inductif, les fonctions “récursives” opérant sur le type *Couleur* permettent simplement la décomposition par cas avec `match`.
Ecrivons pour illustrer le principe de calcul une petite fonction qui donne une valeur à chaque couleur.

```
Fixpoint valeur_couleur (c:Couleur) : nat :=
  match c with
  | pique ⇒ 1
  | coeur ⇒ 2
  | carreau ⇒ 3
  | trefle ⇒ 4
  end.
```

```
Example ex_valeur_couleur1: valeur_couleur coeur = 2.
Proof.
  compute. reflexivity.
Qed.
```

Pour le raisonnement, Coq génère automatique un principe d'induction qui est en fait un terme de nom *Couleur_ind* (dans le cas général, c'est bien sur <`Type`>*_ind* où <`Type`> est le type inductif que l'on considère).
Regardons le type de ce principe inductif.

```
Check Couleur_ind.
```

```
Couleur_ind
     : ∀ P : Couleur → Prop,
       P pique → P coeur → P carreau → P trefle → ∀ c : Couleur, P c
```

Nous pouvons “lire” ce principe inductif de la façon suivante. pour montrer qu'une propriété P portant sur les couleurs est vraie, il suffit de montrer que P est vraie pour chacun des différents cas de couleur.

Ce principe d'induction caractérise le raisonnement par cas sur les couleurs. Les tactiques `destruct` et `induction` ont en fait exactement le même comportement pour un type non-récursif (on utilisera de préférence `destruct` pour insister sur le type de raisonnement effectué).
Voici une illustration de ce principe:

```
Lemma couleur_surj:
  ∀ c : Couleur,
    c = pique ∨ c = coeur ∨ c = carreau ∨ c = trefle.
Proof.
  destruct c. (*  essayer aussi avec induction c *)
  - (*  cas pique *)
    left.
    reflexivity.
  - (*  cas coeur *)
    right.
    left.
    reflexivity.
  - (*  cas carreau *)
    right. right.
    left.
    reflexivity.
  - (*  cas trefle *)
    right. right. right.
    reflexivity.
Qed.
```

1.1.1 Exercice

Montrer :

```
Lemma borne_valeur:
  ∀ c : Couleur,
    (0 < (valeur_couleur c)) ∧ ((valeur_couleur c) ≤ 4).
```

1.2 Types paramétrés

Les types paramétrés non-récursifs sont à peine plus complexes que les types énumérés simples. Prenons l'exemple de figures géométriques encodées de la façon suivante :

```
Inductive geom : Set :=
  | point: nat → nat → geom
  | segment: nat → nat → nat → nat → geom
```

```
| triangle: nat → nat → nat → nat → nat → nat → geom
| nogeom : geom.
```

Les règles de formation correspondantes sont les suivantes :

$$\frac{x : nat \quad y : nat}{point\ x\ y : geom} \qquad \frac{x_1 : nat\ y_1 : nat \quad x_2 : nat\ y_2 : nat}{segment\ x_1\ y_1\ x_2\ y_2 : geom}$$

$$\frac{}{nogeom : geom} \qquad \frac{x_1 : nat\ y_1 : nat \quad x_2 : nat\ y_2 : nat \quad x_3 : nat\ y_3 : nat}{triangle\ x_1\ y_1\ x_2\ y_2\ x_3\ y_3 : geom}$$

Le cas *nogeom* représente l'élément neutre des compositions de figures, au sens de la composition suivante :

```
Fixpoint compose_geom (g1 g2:geom) : geom :=
  match g1 with
  | point x1 y1 ⇒ match g2 with
                   | point x2 y2 ⇒ segment x1 y1 x2 y2
                   | segment x2 y2 x3 y3 ⇒ triangle x1 y1 x2 y2 x3 y3
                   | _ ⇒ nogeom
                   end
  | segment x1 y1 x2 y2 ⇒ match g2 with
                   | point x3 y3 ⇒ triangle x1 y1 x2 y2 x3 y3
                   | _ ⇒ nogeom
                   end
  | _ ⇒ nogeom
  end.

Example ex_segment: compose_geom (point 0 0) (point 1 1) =
  segment 0 0 1 1.
Proof.
  compute. reflexivity.
Qed.
```

Pour raisonner sur les géométries, on dispose encore une fois d'un principe d'induction généré automatiquement.

```
Check geom_ind.
```

```
geom_ind
     : ∀ P : geom → Prop,
       (∀ n n0 : nat, P (point n n0)) →
       (∀ n n0 n1 n2 : nat, P (segment n n0 n1 n2)) →
       (∀ n n0 n1 n2 n3 n4 : nat, P (triangle n n0 n1 n2 n3 n4)) →
       P nogeom → ∀ g : geom, P g
```

1.2.1 Exercice

Montrer :

```
Lemma compose_nogeom:
  ∀ g : geom, compose_geom nogeom g = nogeom.
```

1.3 Types polymorphes

Les types polymorphes non-récursifs sont paramétrés non-pas par des valeurs mais par des types. Comme types et valeurs sont unifiés en Coq, on pourrait croire qu'il n'existe pas de différence avec la catégorie précédente. Philosophiquement, c'est vrai mais en pratique les types polymorphes sont explicités (lorsque cela est possible) au niveau du type inductif lui-même et non de ses constructeurs.
Prenons l'exemple d'un type option "maison".

```
Inductive maybe (A:Set) : Type :=
  Nothing : maybe A
| Just : A → maybe A.

Arguments Nothing [A].
Arguments Just [A] _.
```

On a ajouté ci-dessus quelques annotations qui indiquent à *Coq* que l'argument A du type *maybe* peut généralement être inféré à partir du contexte. Ceci permet d'alléger les notations. Sous forme de règles d'inférences, on peut écrire :

$$\frac{A : Set}{Nothing : maybe\ A} \quad \frac{A : Set \quad a : A}{Just\ a : maybe\ A}$$

Le principe d'induction associé est le suivant.

```
Check maybe_ind.
```

```
maybe_ind
     : ∀ (A : Set) (P : maybe A → Prop),
       P Nothing → (∀ a : A, P (Just a)) → ∀ m : maybe A, P m
```

Considérons une fonction simple permettant d'appliquer une fonction unaire à une donnée de type *maybe*.

```
Definition maybe_map {A B:Set} (f:A→B) : maybe A → maybe B :=
  fun (ma:maybe A) ⇒ match ma with
                          | Nothing ⇒ Nothing
                          | Just a ⇒ Just (f a)
                        end.
```

On peut bien sûr démontrer des propriétés sur cette fonction.

```
Lemma maybe_map_id:
  ∀ A : Set, ∀ ma : maybe A,
    maybe_map (fun a:A ⇒ a) ma = (fun (ma:maybe A) ⇒ ma) ma.
Proof.
  intros A ma.
  unfold maybe_map.
  destruct ma as [|a] ; reflexivity.
Qed.
```

1.3.1 Exercice

Soit la fonction suivante :

```
Definition maybe_compose {A B C : Set} (f:A→B) (g:B→C) : A→C :=
  fun (a:A) ⇒ g (f a).
```

Démontrer :

```
Lemma maybe_map_compose:
  ∀ A B C : Set, ∀ ma : maybe A,
  ∀ f : A→B, ∀ g: B→C,
    maybe_map (maybe_compose f g) ma
    = (maybe_compose (maybe_map f) (maybe_map g)) ma.
```

1.4 Types récursifs simples

Faire du raisonnement par cas sur des énumérations est intéressant mais un peu limité en expressivité. Il est temps de faire de la "vraie" induction.
Le type inductif le plus simple et probablement l'un des plus intéressant est le type *nat* des entiers naturels dans l'axiomatique de Peano. Il s'agit bien sûr d'un type prédéfini mais il ne s'agit pas d'un type primitif.

```
Print nat.
```

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat
```

Les règles d'inférence correspondantes sont les suivantes :

$$\frac{}{O : nat} \qquad \frac{n : nat}{Sn : nat}$$

Le principe inductif généré pour *nat* est le suivant:

```
Check nat_ind.
```

nat_ind
: $\forall\ P : nat \rightarrow$ Prop,
$P\ 0 \rightarrow (\forall\ n : nat,\ P\ n \rightarrow P\ (S\ n)) \rightarrow \forall\ n : nat,\ P\ n$

Il s'agit bien sûr du fameux principe d'induction sur les entiers naturels. Pour montrer qu'une propriété $P(n)$ est vraie pour tout n alors il suffit de montrer:

- que $P(0)$ est vraie
- et qu'en supposant que pour tout n $P(n)$ est vraie, alors $P(n+1)$ est vraie.

Nous avons déjà exploité ce principe d'induction dans le thème arithmétique, donc regardons un second exemple.

```
Inductive list_Couleur : Set :=
  | Nil_Couleur: list_Couleur
  | Cons_Couleur: Couleur → list_Couleur → list_Couleur.
```

Les règles d'inférences correspondantes sont les suivantes :

$$\frac{}{Nil\_Couleur : list\_Couleur} \quad \frac{c : Couleur \quad l : list\_Couleur}{Cons\_Couleur\ c\ l : list\_Couleur}$$

1.4.1 Exercice

Donner le principe d'induction *list_Couleur_ind* associé au type *list_Couleur* (sans faire `Print` ou `Check` *list_Couleur_ind* bien sûr !).

1.4.2 Exercice

On se donne les définitions récursives suivantes :

```
Fixpoint meme_couleur (c : Couleur) (l : list_Couleur) : Prop :=
  match l with
    | Nil_Couleur ⇒ True
    | Cons_Couleur e l' ⇒ (e = c) ∧ meme_couleur c l'
  end.

Fixpoint somme_couleurs (l : list_Couleur) : nat :=
  match l with
    | Nil_Couleur ⇒ 0
    | Cons_Couleur e l' ⇒ (valeur_couleur e) + (somme_couleurs l')
  end.

Fixpoint longueur (l : list_Couleur) : nat :=
  match l with
    | Nil_Couleur ⇒ 0
```

```
    | Cons_Couleur _ l' ⇒ S (longueur l')
  end.
```

Montrer le lemme suivant :

```
Lemma couleur_unique:
  ∀ c : Couleur, ∀ l : list Couleur,
    meme_couleur c l
    → (somme_couleurs l) = (longueur l) × (valeur_couleur c).
```

1.5 Types récursifs polymorphes

Les types récursifs polymorphes représentent la généralisation naturelle des types polymorphes non-récursifs comme *Maybe* et des types monomorphes récursifs comme *list_Couleurs*. Un exemple emblématique des types récursifs polymorphes est le type *list* :

```
Print list.
```

```
Inductive list (A : Type) : Type :=
    nil : list A
 | cons : A → list A → list A
```

Le principe inductif généré pour l'occasion est le suivant :

```
Check list_ind.
```

```
list_ind
     : ∀ (A : Type) (P : list A → Prop),
       P nil →
       (∀ (a : A) (l : list A), P l → P (a :: l)) →
       ∀ l : list A, P l
```

1.5.1 Exercice

Question 1

Définir le type *bintree* des arbres binaires polymorphes avec deux constructeurs :

- *leaf* sans argument
- *node* avec trois arguments : étiquette du noeud, sous-arbre gauche et sous-arbre droit

Le principe d'induction associé doit avoir la forme suivante :

```
Check bintree_ind.
```

```
bintree_ind
     : ∀ (A : Type) (P : bintree A → Prop),
       P leaf →
       (∀ (a : A) (b : bintree A),
        P b → ∀ b0 : bintree A, P b0 → P (node a b b0)) →
       ∀ b : bintree A, P b
```

Question 2

Définir une fonction *nsize* qui retourne la taille d'un arbre binaire en nombre de noeuds internes (sans les feuilles).
Par exemple :

```
Definition bintree_ex1 : bintree nat :=
  (node 1
        (node 2
              (node 3 leaf leaf)
              (node 4
                    (node 5 leaf leaf)
                    (node 6 leaf (node 7 leaf leaf))))
        (node 8
              (node 9 leaf leaf)
              leaf)).

Example nsize_ex1:
  nsize bintree_ex1 = 9.
Proof.
  simpl. reflexivity.
Qed.
```

Question 3

Définir la fonction *lsize* qui compte cette-fois ci le nombre de feuilles d'un arbre binaire.
Par exemple :

```
Example lsize_ex1:
  lsize bintree_ex1 = 10.
Proof.
  compute. reflexivity.
Qed.
```

Démonter le lemme suivant :

```
Lemma node_leaf_size:
  ∀ A : Type, ∀ t : bintree A,
    (lsize t) = S (nsize t).
```

1.5.2 Exercice

Question 1

Définir la fonction *lprefix* qui retourne la liste des étiquettes d'un arbre binaire *t* selon un parcours préfixe.
Par exemple :

```
Example lprefix_ex1:
  lprefix bintree_ex1 = 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: 8 :: 9 :: nil.
Proof.
  compute. reflexivity.
Qed.
```

Question 2

Montrer le lemme suivant :

```
Lemma nsize_length:
  ∀ A : Set, ∀ t : bintree A,
    nsize t = length (lprefix t).
```

1.5.3 Exercice

Question 1

Définir la fonction *bmap* qui effectue un *map* d'une fonction unaire *f* sur un arbre binaire *t*.
Par exemple :

```
Example bmap_ex1:
  bmap (fun (n:nat) ⇒ n + n) bintree_ex1
  = node 2
        (node 4
              (node 6 leaf leaf)
              (node 8
                    (node 10 leaf leaf)
                    (node 12 leaf (node 14 leaf leaf))))
        (node 16 (node 18 leaf leaf) leaf).
```

```
Proof.
  compute. reflexivity.
Qed.
```

Question 2

Démontrer le lemme suivant :

Lemma *map_bmap*:
  $\forall$ A B : Set,
  $\forall$ f : $A \rightarrow B$, $\forall$ t : *bintree* A,
    lprefix (*bmap* f t) = *map* f (*lprefix* t).

1.6 Récursion mutuelle

Les choses se compliquent quelque peu lorsque l'on s'intéresse aux types mutuellement récursifs. Voici une définition pour le type *gentree* des arbres généraux (également appelés *rose trees* dans la littérature) :

Inductive **gentree** (A:Set) : Set :=
| gnode: $A \rightarrow$ **forest** $A \rightarrow$ **gentree** A
with **forest** (A:Set) : Set :=
| fnil: **forest** A
| fcons: **gentree** $A \rightarrow$ **forest** $A \rightarrow$ **forest** A.

Arguments gnode [A] _ _.
Arguments fnil [A].
Arguments fcons [A] _ _.

Les règles d'inférence correspondantes sont les suivantes :

$$\frac{A : Set \quad a : A \quad f : forest\ A}{gnode\ a\ f : gentree\ A}$$

$$\frac{A : Set}{fnil : forest\ A} \quad \frac{A : Set \quad g : gentree\ A \quad f : forest\ A}{fcons\ g\ f : forest\ A}$$

Le principal problème de ce type de définition mutuellement récursive en Coq est que le principe d'induction généré automatiquement n'est pas utilisable. En fait, nous avons besoin d'un principe d'induction combiné pour les noeuds et les forêts mais Coq génère deux principes séparés.

Check gentree_ind.

gentree_ind
    : $\forall$ (A : Set) (P : *gentree* $A \rightarrow$ Prop),

$(\forall\ (a : A)\ (f : forest\ A),\ P\ (gnode\ a\ f)) \rightarrow$
$\forall\ g : gentree\ A,\ P\ g$

```
Check forest_ind.
```

forest_ind
$: \forall\ (A : \texttt{Set})\ (P : forest\ A \rightarrow \texttt{Prop}),$
$P\ (fnil\ A) \rightarrow$
$(\forall\ (g : gentree\ A)\ (f0 : forest\ A),\ P\ f0 \rightarrow P\ (fcons\ g\ f0)) \rightarrow$
$\forall\ f1 : forest\ A,\ P\ f1$

Montrons en pratique ce qu'il se passe si on essaye d'effectuer une démonstration par induction en se basant sur *gentree_ind*.

```
Fixpoint gsize {A:Set} (t:gentree A) : nat :=
  match t with
    | gnode _ f => S (fsize f)
  end
with fsize {A:Set} (f:forest A) : nat :=
       match f with
         | fnil => 0
         | fcons e f' => (gsize e) + (fsize f')
       end.

Fixpoint lgprefix {A:Set} (t:gentree A) : list A :=
  match t with
    | gnode e f => e::(lfprefix f)
  end
with lfprefix {A:Set} (f:forest A) : list A :=
     match f with
       | fnil => nil
       | fcons e f' => (lgprefix e) ++ (lfprefix f')
     end.

Lemma gsize_length:
  ∀ A : Set, ∀ t : gentree A,
    gsize t = length (lgprefix t).
Proof.
  intros A t.
  induction t.
  simpl.
  Abort.
```

Le dernier but à prouver est le suivant :

```
 A : Set
  a : A
```

```
  f : forest A
  ============================
   S (fsize f) = S (length (lfprefix f))
```

Or, on ne possède aucune hypothèse d'induction sur la forêt. Pour obtenir un principe d'induction combiné, on utilise la commande `Scheme` de Coq.

```
Scheme gentree_ind' :=
  Induction for gentree Sort Prop
  with forest_ind' :=
    Induction for forest Sort Prop.
Check gentree_ind'.
```

```
gentree_ind'
     : ∀ (A : Set) (P : gentree A → Prop) (P0 : forest A → Prop),
       (∀ (a : A) (f : forest A), P0 f → P (gnode a f)) →
       P0 fnil →
       (∀ g : gentree A,
        P g → ∀ f1 : forest A, P0 f1 → P0 (fcons g f1)) →
       ∀ g : gentree A, P g
```

Le principe d'induction généré combine correctement les inductions de noeud et les inductions de forêts.
Reprenons notre démonstration. La principale difficulté est que Coq ne peut découvrir automatiquement le principe d'induction à appliquer en pratique. La tactique `induction` ne fonctionne donc pas et nous devons exploiter la tactique de plus bas niveau `elim` en indiquant explicitement le prédicat *P0* dans le principe d'induction ci-dessus. Le rôle de ce prédicat est de propager le but à prouver (ici, que la taille est égale à la longueur du préfixe) sur les forêts.

```
Lemma gsize_length:
  ∀ A : Set, ∀ t : gentree A,
    gsize t = length (lgprefix t).
Proof.
  intros A t.
  elim t using gentree_ind' with
  (P0:= fun f:forest A ⇒ fsize f = length (lfprefix f)).
  - (*  cas des noeuds *)
    intros a f H1.
    simpl.
    rewrite H1.
    reflexivity.
  - (*  cas des forêts vides *)
    simpl.
```

```
    reflexivity.
  - (*  cas des forêts non-vides *)
    intros g Hg f Hf.
    simpl.
    rewrite Hg.
    rewrite Hf.
    (*  SearchRewrite (length (_ ++ _))
       app_length:
         forall (A : Type) (l l' : list A),
             length (l ++ l') = length l + length l' *)
    rewrite app_length.
    reflexivity.
Qed.
```

Remarque : on serait tenté d'utiliser le type *list* pour les forêts, ce qui permettrait de réutiliser la bibliothèque de fonctions et de lemmes sur les listes. C'est bien sûr possible mais en fait non-trivial en Coq. La difficulté concerne la génération du principe d'induction combiné. En fait la commande `Scheme` ne fonctionne plus et il faut créer le principe d'induction de façon complètement manuelle.

1.6.1 Exercice

Question 1

Définir la fonction *gmap* qui effectue un *map* d'une fonction unaire f sur un arbre général t.

Question 2

Démontrer le lemme suivant :

```
Lemma map_gmap:
  ∀ A B : Set,
  ∀ h : A→B, ∀ t : gentree A,
    lgprefix (gmap h t) = map h (lgprefix t).
```

1.7 Types dépendants

Pour enrichir l'expressivité de `Inductive`, au-delà de la définition de types "à la ML", consiste à permettre les dépendances entre types et valeurs.
Du point de vue du calcul, la possibilité de faire dépendre les types de valeurs permet de proposer un mode de programmation où la logique et le calcul s'entremêlent. Cela ouvre notamment la porte à la programmation certifiée. Le principe, grossièrement, est de permettre la vérification de contraintes sémantiques par typage au moment de la compilation. On le

verra lors du prochain thème, mais cela permet également un mode de programmation “à la prolog” à l’ordre supérieur.
Commençons par un exemple jouet glané sur le web : les dominos.

```
Inductive domino : nat → nat → Type :=
  | block: ∀ m n, domino m n
  | chain: ∀ m n p, domino m n → domino n p → domino m p.
Arguments chain [m n p] _ _.
```

Le type *domino* correspond à une généralisation du jeu classique, dans lequel on ne peut composer deux dominos que s’ils ont un nombre en commun.
Voici par exemple une construction correcte :

```
Eval compute in (chain (chain (block 3 8) (block 8 4)) (block 4 9)).

    = chain (chain (block 3 8) (block 8 4)) (block 4 9)
    : domino 3 9
```

En revanche, si on casse la contrainte de construction, une erreur de typage est signalée.

```
Eval compute in chain (chain (block 3 8) (block 7 4)) (block 4 9).
—
Error: The term "block 7 4" has type "domino 7 4"
 while it is expected to have type "domino 8 ?600".
```

Ici Coq signale précisément l’erreur. Cet exemple montre qu’on atteint ici une sorte de *Graal* de la programmation sûre : détecter les erreurs sémantiques à la compilation. Même si cette vision est assez simplificatrice, on peut dire que les langages à types dépendants comme notamment *Idris* sont d’ores et déjà utilisables pour l’expérimentation.

D’un point de vue pratique, un grand classique des types dépendants est le type *vecteur* ou liste à longueur spécifiée. Les vecteurs sont fournis dans la bibliothèque standard de Coq avec le module *Vector* mais nous allons ici les reconstruire.
La définition proposée est la suivante :

```
Inductive vector (A:Set) : nat → Set :=
  | vNil : vector A O
  | vCons : ∀ n, A → vector A n → vector A (S n).
Arguments vNil [A].
Arguments vCons [A][n] _ _.
```

La dépendance valeur/type est claire sur les règles d’inférences correspondantes :

$$\frac{A : Set}{vNil : vector\ A\ O} \qquad \frac{A : Set \quad n : nat \quad a : A \quad v : vector\ A\ n}{vCons\ a\ v : vector\ A\ (Sn)}$$

Un vecteur de type *vector* est donc soit le vecteur vide *vNil* de taille 0 soit un vecteur de taille $n+1$ résultat de l'adjonction d'un premier élément à un vecteur reste de taille n.
Le principe d'induction associé ne pose pas de problème particulier.

```
Check vector_ind.
```

```
vector_ind
     : ∀ (A : Set) (P : ∀ n : nat, vector A n → Prop),
       P 0 (vNil A) →
       (∀ (n : nat) (a : A) (v : vector A n),
        P n v → P (S n) (vCons a v)) →
       ∀ (n : nat) (v : vector A n), P n v
```

Pour illustrer l'utilisation des vecteurs, considérons la fonction *vapp* de concaténation de deux vecteurs.

```
Fixpoint vapp {A:Set} {n1:nat} (v1:vector A n1) {n2:nat} (v2:vector A n2) : vector A
(n1 + n2) :=
  match v1 in (vector _ n1) return (vector A (n1 + n2)) with
    | vNil ⇒ v2
    | vCons _ e v1' ⇒ vCons e (vapp v1' v2)
  end.
```

On voit ici une différence importante avec la programmation fonctionnelle classique, puisque l'on réalise effectivement des calculs dans les types.

```
Example vapp_ex1:
  vapp (vCons 1 (vCons 2 (vCons 3 vNil)))
       (vCons 4 (vCons 5 vNil))
  = vCons 1 (vCons 2 (vCons 3 (vCons 4 (vCons 5 vNil)))).
Proof.
  compute.
  reflexivity.
Qed.
```

Contrairement aux listes “classiques”, la définition de la longueur est triviale avec les vecteurs.

```
Definition vlength {A:Set} {n:nat} (v:vector A n) : nat := n.
```

Ceci nous permet d'effectuer une première démonstration simple.

1.7.1 Exercice

Démontrer le lemme suivant :

```
Lemma vappa_vlength:
  ∀ A : Set,
  ∀ n1 n2 : nat,
```

```
  ∀ v1 : vector A n1,
  ∀ v2 : vector A n2,
    vlength (vapp v1 v2) = n1 + n2.
```

1.7.2 Exercice

Définition la fonction *vmap* permettant d'appliquer une fonction unaire sur un vecteur.
Par exemple :

```
Example vmap_ex1:
  vmap (fun b:bool ⇒ match b with
                      | true ⇒ 1
                      | false ⇒ 0
                    end) (vCons true (vCons false vNil))
  = vCons 1 (vCons 0 vNil).
Proof.
  compute. reflexivity.
Qed.
```

1.7.3 Exercice

Définir une fonction *list_from_vect* permettant de convertir un vecteur en liste.
Par exemple :

```
Example list_from_vect_ex1:
  list_from_vect (vCons true (vCons false vNil)) = true :: false :: nil.
Proof.
  compute. reflexivity.
Qed.
```

1.7.4 Exercice

Démontrer le lemme suivant :

```
Lemma vmap_map:
  ∀ A B:Set,
  ∀ f : A→B,
  ∀ n:nat,
  ∀ v : vector A n,
    list_from_vect (vmap f v) = map f (list_from_vect v).
```

1.7.5 Exercice

Définir la fonction *vect_from_list* de conversion inverse, qui consiste à générer un vecteur à partir d'une liste spécifiée.
Par exemple :

```
Example vect_from_list_ex1:
  vect_from_list (true::false::nil) = vCons true (vCons false vNil).
Proof.
  compute. reflexivity.
Qed.
```

1.7.6 Exercice

Démontrer le théorème suivant.

```
Theorem vect_list_convert:
  ∀ A : Set,
  ∀ l : list A,
    list_from_vect (vect_from_list l) = l.
```