Vers une implémentation de l'algèbre relationnelle en CoQ

Olivier Pons

29 septembre 1995

Table des matières

Chapitre 1

Généralités

1.1 Introduction

Un problème majeur en programmation est d'être sûr que les programmes que l'on conçoit et que l'on utilise accomplissent bien ce que l'on attend d'eux. Généralement on teste le programme sur de nombreux exemples pour se convaincre qu'il fait bien ce que l'on attend de lui, mais cela réserve parfois de mauvaises surprises par la suite. Pour remédier à ce problème, on a conçu des systèmes d'aide à la preuve qui permettent de spécifier les programmes et de les prouver; c'est à dire, montrer par un raisonnement logique que le comportement induit par la spécification est bien celui que suit le programme. On obtient ainsi des programmes dit *certifiés*.

Rappelons brièvement ce que l'on entend par spécifier un programme; c'est donner sous une forme explicite son comportement, donc ce qu'il doit faire mais sans dire comment le faire [1].

On exprimera donc quelles sont les données, quelles sont leurs propriétés, quels sont les résultats et quelles propriétés ils vérifient. On verra par exemple la spécification d'un algorithme réalisant l'union de deux ensembles. Il a comme entrées deux ensembles E_1 et E_2 et calcule en sortie un ensemble E_3

[1] Il faut noter que si cette définition est globalement exacte, dans certains cas (pour l'addition des entiers, ou la concaténation des listes par exemple), une définition équationelle simple est beaucoup plus naturelle qu'une spécification logique.

vérifiant la propriété

$$\forall x, ((x \in E_1 \vee x \in E_2) \leftrightarrow (x \in E_3))$$

Remarquons qu'en général une spécification peut être réalisée par plusieurs programmes. Prenons par exemple les nombreux algorithmes de tri, leur spécification prend toujours en entrée une liste quelconque et calcule en sortie une liste ordonnée.

La motivation de ce travail a été d'obtenir en utilisant le système Coq, une implémentation certifiée de l'algèbre relationnelle des bases de données.

Nous n'avons pas choisi une représentation des ensembles et des relations dans le style des types de données abstraits telle qu'on les utilise dans les spécifications algébrique mais, au contraire, une représentation concréte des ensembles et des relations, en terme de listes et de vecteurs.

Après un bref rappel de quelques notions essentielles de Coq, nous étudions donc dans un premier temps les structures concrètes que sont les listes et les vecteurs; puis nous les utilisons pour implémenter les ensembles et les relations. On peut ensuite étudier la spécification des opérations de l'algèbre relationnelle.

1.2 Brève introduction à Coq

La logique sous-jacente au systéme Coq est un lambda calcul typé, le Calcul des Constructions Inductives (C.I.C). C'est un lambda calcul d'ordre supérieur, avec types dépendants et avec définitions inductives.

Dans ce système, prouver une formule ϕ sous les hypothèses $H_1 \ldots H_k$, c'est trouver un λ -terme t tel que, l'on ai le jugement de typage

$$x_1 : H_1, \ldots x_k : H_k \vdash t : \phi$$

(ce que l'on lit t est de type ϕ dans le contexte où x_1 est de type H_1, ..., x_k est de type H_k)

Du point de vu programation, on peut dire que t est un programme qui réalise la specification ϕ.

Mais tel quel, ce programme peut être totalement inefficace du fait que en plus de l'algorithme explicite de réalisation de la spécification, il contient de nombreuses informations logiques annexes. C'est pour arriver à déduire d'une

preuve un programme plus efficace que l'on a introduit le concept d'extraction [2] . Cela consiste à éliminer de la preuve toute les informations qui n'ont pas un contenu calculatoire. On définit pour cela une fonction d'extraction ou fonction d'oubli, qui éliminera toutes les parties logiques d'une preuve pour ne conserver que les parties calculatoires.

Ainsi, dans le C.I.C, d'une preuve d'une proposition π

$$(x : A)\{y : B|P(x, y)\}$$

on peut extraire un programme fonctionnel satisfaisant la spécification π, *i.e.* qui à tout objet x de type A associe un objet y de type B tel que $P(x, y)$ soit vérifié.

Pour mettre en oeuvre ce principe d'extraction, il est nécessaire de pourvoir distinguer les termes qui ont un contenu calculatoire de ceux qui n'en on pas. Pour cela on a introduit deux sortes de base dans notre lambda-calcul typé :

- (Set:TypeSet) Pour représenter les termes à contenu calculatoire
- (Prop:Type) Pour les termes logiques

1.3 Quelques points importants

On développe ci dessous un choix non exhaustif de notions qui ont étés importantes pour notre travail.

1.3.1 La notion de définition inductive

Habituellement, un ensemble défini inductivement est le plus petit ensemble vérifiant un certain nombre d'axiomes. Par exemple l'ensemble des entiers naturels peut être vu comme le plus petit ensemble N vérifiant

- $O \in N$ (O est un entier)
- $si\ n \in N\ alors\ n + 1 \in N$ (l'ensemble est clos par successeur)

[2] La référence classique sur le sujet est [Paulin-Mohring89], mais le chapitre 2 de [Parent95] constitue une excellente introduction.

Coq permet ce genre de définition, pour cela on dispose du mot clef `Inductive` On définie donc les entiers par:

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

Ce qui construit effectivement les objets `nat:Set, O:nat , S:nat->nat`, mais donne aussi les schémas de récurrences (ou d'élimination) suivant: `nat_ind, nat_rec,` etc ..., qui correspondent aux éliminations vers les differents types. Pour les entiers, le schéma d'élimination vers les `Set` est :

```
nat_rec
     : (P:nat->Set)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

Cela permet donc le raisonnement par récurrence sur les objets construits inductivement. Pour montrer une propriété `P` sur les entiers il suffira de montrer que `(P O)` est vérifié, et si `(P (S n))` est vérifié sous la condition `(P n)`, alors tout entier vérifie `P`.

De plus ces schémas de récurences ne sont pas seulement des axiomes logiques, ils ont aussi un comportement calculatoire qui permettra de définir des fonctions pour manipuler les objets définis inductivement. Ils permettent par exemple d'écrire l'addition sur les entiers par:

```
Definition fun_add := [n,m:nat]<nat>Match n with
                                    m
                                    ([p:nat][n':nat](S n'))
                                    end.
```

1.3.2 Définition fonctionnelle et définition "à la Prolog"

La richesse de Coq fait que l'on a souvent plusieurs manières d'exprimer la même idée. Ainsi toutes les fonctions peuvent être définies directement comme un lambda-terme, ou "à la Prolog", c'est à dire sous forme de prédicat inductif.

On a donné précédemment la définition fonctionnelle de l'addition sur les entiers.

"À la Prolog", la fonction d'addition sur les entiers s'exprime par le prédicat:

```
Inductive Prop_add :nat->nat->nat->Prop :=
ad_O :(n:nat)(Prop_add O n n)
|ad_s :(n1,n2,n3:nat )
            (Prop_add n1 n2 n3)->(Prop_add (S n1) n2 (S n3)).
```

où (`(Prop_add n1 n2 n3) signifie que n3 est la somme de n1 est n2`)

En pratique nous préfererons utiliser la version fonctionnelle car la forme prédicative oblige à manipuler constamment trois entiers et semble donc plus lourde à manipuler. Cela est encore plus vrai sur des fonctions plus compliquées.

Notons enfin que le terme fonctionnel peut être déduit de la version prédicative en prouvant (par induction sur `n1`) le lemme suivant:

```
Lemma exist_add :(n1,n2:nat){n3:nat | (Prop_add n1 n2 n3)}.
```

1.3.3 Realizer et program

On a parlé plus haut du processus d'extraction des termes du C.I.C vers $\mathcal{F}_{\omega}^{idt}$(c'est la synthèse de programme). La réciproque de cela consiste à partir d'un programme et d'une spécification pour retrouver la preuve de cette dernière à partir du programme. On parle alors de synthèse de preuve. C'est l'objet de la tactique `Program`.

Regardons la différence entre les deux approches sur l'exemple simple de la fonction prédécesseur:

L'approche "classique", synthèse de programme: On donne une spécification du prédécesseur :

```
Lemma pred_spec :(n:nat){p:nat |(n=(S p))}+{n=O}.
```

On peut la prouver grâce aux tactiques usuelles de Coq soit

```
Induction n.  (* indique un raisonement par recurrence sur n *)
   Right.                 (* le cas de base n=O *)
   Try Trivial.

   Intros.                 (* l'etape de recurrence *)
```

```
   Left.
   Exists n0.
   Try Trivial.

Defined.
```

ensuite la commande `Extraction` permet d'extraire de cette preuve la fonction calculant le prédécesseur d'un entier.

```
Coq < Extraction pred_spec.
pred_spec ==>[n:nat]
               (nat_rec (sumor nat) (inright nat)
                 [n0:nat][_:(sumor nat)] (inleft nat n0) n)
 :  nat->(sumor nat)
```

L'approche synthèse de preuve et l'utilisation de `Realizer` et `Program`

On construit une fonction sensée calculer le prédécesseur d'un entier:

```
Definition pred1 := [n:nat]<(Exc nat)>Case n of (error nat)
                              ([u:nat](value nat u))
                              end
```

On peut montrer qu'elle vérifie bien notre spécification, de la façon suivante: :

```
Lemma pred_spec :(n:nat){p:nat |(n=(S p))}+{n=O}.
Realizer pred1.
Program_all.

Save.
```

`Realizer` se charge de vérifier que le type du programme et la valeur informative du terme extrait du but coincident. Puis `Program` tente d'engendrer la preuve en se servant de la structure du programme; à défaut, il engendre un certain nombre de sous-buts qu'il faudra encore prouver.

Introduction de la notion d'exception, le type summor Dans le paragraphe précédent on a utilisé la notion d'exception qui sera beaucoup utilisée dans la suite. On a un `type Exc`, défini par:

```
Inductive Exc [A:Set] : Set :=
               value : A->(Exc A)
               | error : (Exc A)
```

Le type `sumor` qui, apparaît dans la fonction extraite, a été introduit pour permettre de gérer les exceptions. C' est un type inductif defini par :

```
Inductive sumor [A:Set;B:Prop] : Set :=
     inleft : A->(sumor A B)   (* cas ou on renvoie la valeur *)
    | inright : B->(sumor A B) (* cas des exceptions *)
```

Lors de l'extraction, on l'a dit, les objets de type `Prop` sont éliminés. il reste donc:

```
Coq < Extraction sumor.
sumor ==> Inductive sumor [A:Set] : Set :=
              inleft : A->(sumor A)
            | inright : (sumor A)
```

On a donc plonger A dans un ensemble plus vaste qui contient A et la valeur qui sera celle des exceptions.

1.4 Les problèmes de décidabilité

Lorsque l'on manipule des relations, par exemple l'égalité, on a souvent besoin d'un programme permettant de décider si cette relation est verifiée ou non.

On utilise pour cela le type `sumbool` défini par

```
Inductive sumbool [A:Prop;B:Prop] : Set :=
    left : A->(sumbool A B)
 | right : B->(sumbool A B)
```

(mais on dispose de la syntaxe `{A}+{B}`qui est un affichage agréable pour `(sumbool A B)`)

Ce type est une version calculatoire (puisqu' il renvoie un `Set`) de la disjonction logique. Par exemple, d'une preuve de `(x,y:nat){x=y}+{ ~ x=y}`, on peut extraire un programme qui teste l'égalité sur les entiers

Chapitre 2

Les structures concrètes

Dans tout ce chapitre les objets que nous définirons seront paramétrés par un ensemble de base, `s:Set`. De plus on affirme que l'égalité sur ce type de base est décidable. Ce que l'on exprime par l'axiome suivant:

```
Axiom eqs_dec : (a:s)(b:s){a=b}+{~a=b}
```

2.1 Les listes

2.1.1 Rappel de la définition inductive

```
Inductive list : Set :=
    nil : list
   | cons : s->list->list
```

On définit ensuite les outils classiques de manipulation des listes.

2.1.2 Quelques fonctions de base

La concaténation

Comme nous l'avons dit toutes les opérations peuvent être définies sous forme fonctionnelle ou de manière prédicative ("à la Prolog"). Pour la concaténation la définition fonctionnelle est :

```
Fixpoint fun-app [l:list] : list -> list
```

```
      := [m:list ]<list>Case l of
                (* nil *) m
           (* cons a l1 *) [a:s][l1:list](cons a (fun-app l1 m)) end.
```

tandis que la définition prédicative est donnée par:

```
Inductive prop-app :list ->list ->list ->Prop :=
ap_l_nil :(l2:list)(prop-app nil l2 l2)
|ap_l_cons :(l1,l2,l3:list )(x:s)(prop-app l1 l2 l3)
                  ->(prop-app (cons x l1) l2 (cons x l3)).
```

Cette deuxième définition est plus lourde à manipuler puisqu'elle oblige à garder constamment trois listes et l'hypothèse `(prop-app l1 l2 l3)`. Nous utiliserons donc par la suite des définitions fonctionnelles.

On peut évidemment prouver que la fonction `fun-app` vérifie le prédicat `prop-app`, en montrant le lemme suivant:

```
Lemma valide_app_list :
 (l1,l2:list)
       (prop-app l1 l2 (fun-app l1 l2)).
```

On démontre également quelques propriétés fondamentales de la concaténation :

- L'associativité

  ```
  Lemma app_ass :
   (l,m,n : list)
         (fun-app(fun-app l m) n)=(fun-app l (fun-app m n)).
  ```

- Concaténer une liste `l` avec la liste vide renvoie la liste `l`

  ```
  Lemma fun-app_nil_end : (l:list)(l=(fun-app l nil)).
  ```

- etc ...

Les tests de nullité

On aura besoin pour la suite, du prédicat `Isnil` qui exprime qu'une liste est vide:

```
Definition Isnil : list -> Prop := [l:list]<list>nil=l.
```

On montre trivialement que `nil` vérifie ce prédicat mais que `cons` ne le vérifie pas.

On spécifie ensuite, une fonction permettant de décider si une liste est vide ou non. La preuve de cette spécification est un programme qui renvoie le booléen `true` quand la liste est vide, `false` sinon.

```
Lemma Null : (l:list){(Isnil l)}+{~(Isnil l)}.
Realizer [l:list]<bool>Case l of
                       (* nil *) true
                  (* cons a m *) [a:s][m:list]false
 end.
Program_all.
Save.
```

({A}+{B} est une écriture agréable pour `(sumor A B)`; le type `sumor` etant une version calculatoire (car il renvoie un objet de type `Set`) de la disjonction logique.)

Les fonction Head et Tail

Nous définissons maintenant la fonction `tail` qui renvoie la queue d'une liste:

```
Definition tail := [l:list ]<list >Case l of
           (* nil *)        nil
           (* cons s a m *) [a:s][m:list]m end
:list->list.
```

On peut donner une spécification de la notion de queue d'une liste en disant : "pour toute liste `l`, on peut construire une liste `m` telle que soit il existe un élément `a` (que l'on ne cherche pas à construire),soit `l` et `m` sont

vides". On s'assure ensuite via `Realizer` et `Program_all` que la fonction tail que nous avons definie vérifie bien cette spécification.

```
Lemma Tl : (l:list){m:list|(Ex [a: s] (cons a m)=l)
                          \/ ((Isnil l) /\ (Isnil m)) }.
Realizer tail.
Program_all.
Left; Exists a; Auto.
Save.
```

On voit sur cette dernière spécification qu'il y a en Coq deux façons d'exprimer l'existence d'un élément vérifiant une propriété donnée; l'une constructive ("extractible"), c'est à dire qui nous permet de récupérer l'élément `m` dont on parle, l'autre purement logique qui permet juste d'exprimer l'existence d'un élément `a` qui ne nous intéresse pas en tant que tel.

La définition de la fonction `head`, qui renvoie la tête d'une liste, soulève le problème de la gestion des exceptions [1], on en donne deux versions; la première prend un argument supplémentaire qui sera renvoyé dans le cas qui pose problème (c'est à dire si la liste en entrée est vide).

```
Definition head  :=[erreur_list_vide:s][l:list ]<s>Case l of
           (* nil *)       erreur_list_vide
           (* cons s a m *) [a:s][m:list]a end
 :s->list->s.
```

La seconde utilise l'implémentation des exceptions. c'est une preuve de la spécification suivante:

"pour toute liste `l`, soit on peut trouver un élément `a` tel qu'il existe une liste `m` (qu'on ne cherche pas à construire), vérifiant `(cons a m)=l`, soit la liste est vide".

Ce qu'on vérifie grâce à `Realizer` et `Program`:

```
Lemma Hd :
```

[1] Dans le cas de la fonction `tail`, on aurait pu renvoyer une exception dans le cas de la liste vide; mais faire le choix de renvoyer la liste vide ne pose pas de problème de typage. Pour `head`, ce que l'on renvoie est un élément de type `s` mais dans le cas ou la liste vide a été donnée en entrée on a aucun objet de type `s` à renvoyer; On ne peut pas renvoyer la liste vide comme beaucoup d'implémentation de *Scheme* car la fonction serait mal typée.

```
   (l:list){a : s | Ex([m: list]<list>(cons a m)=l)}+{Isnil l}.
Realizer [l:list]<(Exc s)>Case l of
                      (* nil *) (error s)
                 (* cons a m *) [a:s][m:list](value s a)
 end.
Program_all.
Exists m; Auto.
Save.
```

Remarquons qu'on peut aussi prouver la spécification sans utiliser `Realizer` mais avec les autres tactiques CoQ (correspondant aux règles logiques) puis en extraire la fonction par la commande `Extraction`.

Ensuite, on prouve que pour toute liste non vide `l`, il existe un élément `a` et une liste `m` tel que `(cons a m)=l`. On a alors une fonction pour les construire. Une preuve de ce lemme est donc une fonction qui prend une liste en entrée et qui renvoie un couple, contenant la tête et la queue de la liste, si elle n'est pas vide, qui lève une exception sinon:

```
Lemma Uncons :
(l:list){a : s & { m: list | <list>(cons a m)=l}}+{Isnil l}.

Realizer [l:list]<(Exc s*list)>Case l of
           (* nil *) (error s*list)
        (* cons a m *) [a:s][m:list](value s*list <s,list>(a,m))
 end.
Program_all.
Save.
```

La longeur d'une liste

On introduit maintenant la notion de longueur. La fonction calculant la longueur d'une liste est :

```
Fixpoint length [l:list] : nat
   := (<nat>Case l of
             (* nil *) O
          (* cons a m *) [a:s][m:list](S (length m)) end).
```

Puis on définit la relation `lel` qui exprime qu'une liste `l` est inférieure (ou égale) à une liste `m` si la longueur de `l` est inférieure (ou égale) à celle de `m` (`le` est la relation $\leq$ des entiers naturels).

```
Definition lel [l,m:list](le (length l) (length m)).
```

On montre que cette relation est un pré-ordre, c'est a dire:
qu' elle est réflexive

```
Lemma lel_refl : (lel l l).
```

et transitive

```
Lemma lel_trans : (lel l m)->(lel m n)->(lel l n).
```

L'appartenance

On continue en définissant une notion fondamentale: l'appartenance. Là encore on donnera deux définitions pour l'appartenance, l'une fonctionnelle :

```
Fixpoint fun_In [a:s;l:list] : Prop :=
        (<Prop>Case l of
       (* nil *) False
     (* Cons b r *)([b:s][m:list](b=a)\/(fun_In a m)) end).
```

l'autre sous forme inductive

```
Inductive Prop_In [a:s]:list ->Prop :=
                       In_hd :(l:list)(In a (cons a l))
                      |In_tl :(b:s)(l:list)(In a l)
                                      ->(In  a (cons b l)).
```

Les deux notions sont équivalentes, comme le montrent les deux lemmes suivants.

```
Lemma fun_In_Prop_In :
     (l:(list s))(a:s)(fun_In s a l)->(Prop_In s a l).

Lemma Prop_In_fun_In :
     (l:(list s))(a:s)(Prop_In s a l)->(fun_IN s a l).
```

Selon les preuves que l'on fera, il sera préférable d'utiliser une définition plutôt que l'autre. L'approche fonctionnelle évitant d'avoir trop souvent recourt aux tactiques d'inversion.

On vérifie les propriétés "classiques" de l'appartenance à une liste :

- *si* $a \in l,\ \forall x,\ a \in x :: l$

```
Lemma fun_In_cons :
     (a,b:s)(l:list)(fun_In b l)->(fun_In b (cons a l)).
```

- Il n'y a rien dans la liste vide

```
Lemma fun_In_nil :(a:s) ~(fun_In a nil).
```

- si `a` appartient a la concaténation de deux listes, alors il appartient à l'une ou à l'autre

```
Lemma fun_In_app_or :
     (l,m:list)(a:s)
   (fun_In a (fun-app l m))->((fun_In a l)\/(fun_In a m)).
```

- etc ...

On introduit le prédicat inductif `AllS` qui permet de parler des listes dont tous les éléments vérifient une propriété donnée.

```
Inductive AllS [P:s->Prop] : list -> Prop
   := allS_nil : (AllS P nil)
   | allS_cons : (a:s)(l:list)(P a)->
                        (AllS P l)->(AllS P (cons a l)).
```

Il permet, par exemple, d'exprimer le fait que tous les éléments d'une liste `l` sont différents d'un élément `a` donné en posant : `(AllS s [b:s]~b=a l)`.

Cela permet de poser et de prouver le lemme suivant:

```
Lemma mem :(a:s)(l:(list s)){(In s a l)}+{(AllS s [b:s]~b=a l)}
```

Ce qui exprime le fait que, soit un élément est dans une liste, soit il est différent de tous les éléments de la liste. Cela nous sera très utile par la suite car on a maintenant une notion de décidabilité sur l'appartenance. Elle pourra être utilisée dans des structures conditionnelles. Cela nous permettra plus loin d'écrire les fonctions sur les ensembles.

On prouve par induction, que si tous les éléments de `l` sont différents de `a`, alors `a` n'est pas dans `l`; soit le lemme suivant:

```
Lemma P1_1 : (a:s)(l:(list s))
     (AllS s [b:s]~b=a l) ->  ~(Inf s a l).
```

2.1.3 Autres notions utiles

On introduit maintenant les notions qui permettront d'exprimer les spécificités des ensembles.

Le nombre d'occurrences:

La notion de nombre d'occurrences est définie inductivement par:

```
Inductive nocc [s:Set] :s->(list s)->nat->Prop :=
noccnil : (a:s)(nocc s a (nil s) O)
|nocc_cons1 : (a:s)(l:(list s))(n:nat)(nocc s a l n)->
              (nocc s a (cons s a l) (S n))
|nocc_cons2 : (a,b:s)(l:(list s))(n:nat)(nocc s a l n)->
              (~(b=a))->(nocc s a (cons s b l) n).
```

On prouve quelques lemmes sur cette notion:

- Si la liste est vide, pour tout élément `a`, le nombre d'occurrences de `a` est zéro:

  ```
  Lemma nocc_nil_O :(n:nat)(a:s)(nocc s a (nil s) n)->(n=O).
  ```

- Réciproquement, si le nombre d'occurrences est zéro, alors la liste est vide

  ```
  Lemma nocc_O_nil :
           (l1:(list s))((a:s)(nocc s a l1 O))->(l1 = (nil s)).
  ```

– Pour tout élément `a` et tout entier `n`, on peut construire une liste qui contient `n` fois `a`.

```
Lemma noc_ex1_a:
      (a:s)(n:nat){l:(list s)| (nocc s a l n)}.
```

– Pour tout élément `a` et toute liste `l` on peut trouver un entier `n` qui est le nombre d'occurrences de `a` dans `l`

```
Lemma nocc_ex_n :
       (a:s)(l:(list s)){n:nat | (nocc s a l n)}.
```

– etc ...

La notion de permutation:

On aura aussi besoin de la notion de permutation définie à partir de la précédente en disant que si une liste est une permutation d'une autre alors tous les éléments présents dans l'une, le sont également dans l'autre et avec le même nombre d'occurrences.

```
Definition Permutation :=
        [s:Set] [l1,l2 :(list s)]
        (n:nat)(a:s)((nocc s a l1 n)<->(nocc s a l2 n)).
```

On montre que c'est une relation d'équivalence.

```
Lemma Permut_sym :
  (l1,l2:(list s))
      (Permut s l1 l2)->(Permut s l2 l1).

Lemma Permut_trans :
 (l1,l2,l3:(list s))
      (Permut s l1 l2)->(Permut s l2 l3)->(Permut s l1 l3).

Lemma Permut_ref :(l1:(list s))(Permut s l1 l1).
```

La seule permutation de la liste vide est la liste vide:

```
Lemma Permut_nil :
   (l:(list s))(Permut s l (nil s))->l=(nil s).
```

Le Prédicat "sans doublet":

Enfin on a introduit le prédicat `sans_d` c'est à dire la notion "d'être sans doublet" qui permettra de représenter les ensembles. Pour cela nous disons qu'une liste est sans doublet si tout élément apparaît au plus une fois [2].

```
Definition sans_d :=
      [s:Set][l:(list s)]
      (a:s)((nocc s a l (S O))\/(nocc s a l O)).
```

On a quelques lemmes liés à cette notion:

- la liste vide est `sans doublet`

  ```
  Lemma sans_d_nil : (sans_d s (nil s)).
  ```

- si une liste est `sans doublet`, la queue de cette liste l'est aussi

  ```
  Lemma sans_d_cons :
     (l:(list s))(x:s)(sans_d s (cons s x l))->(sans_d s l).
  ```

- une permutation d'une liste `sans doublet` est `sans doublet`:

  ```
  Lemma Permute_sans_d :
   (l,l2:(list s))(sans_d s l)->(Permut s l l2)->(sans_d s l2).
  ```

- etc ...

On a aussi besoin de lemmes pour faire le lien entre les notions de nombre d'occurences et d'appartenance. Par exemple le lemme suivant stipule que si le nombre d'occurences d'un élément `a` dans une liste `l` est différent de `O`, alors, `a` appartient à `l`.

```
Lemma nocc_in : (x:s)(l:(list s))~(nocc s x l O)->(fun_In s x l).
```

On prouve aussi la réciproque:

```
Lemma In_nocc : (x:s)(l:(list s))(fun_In s x l)->~(nocc s x l O).
```

[2] *i.e.* pour tout x le nombre d'occurrence dans l est O ou 1

2.2 Les vecteurs

Dans cette partie, on introduira les outils de représentation des t-uplets. Le plus simple étant de définir un type vecteur ou liste de longueur n.

2.2.1 La définition inductive

On introduit donc un nouveau type, les vecteurs. On le définit inductivement par:

```
Inductive Tab :nat->Set:=
        Tnil :(Tab O)
        |Tcons : (n:nat)s->(Tab n)->(Tab (S n)).
```

Et comme dans toutes les définitions inductives, le système génère trois principes d'inductions `Tab_ind, Tab_rec, Tab_rect`. Dès le début, deux problèmes principaux se présentent du fait de l'utilisation **des types dépendants** (*i.e.* qui expriment des propriétés sur les termes, ici le fait d'être de longueur donnée.)

2.2.2 Les problèmes liés à l'égalité sur les types dépendants

On rencontre d'abord des problèmes avec la notion d'égalité sur les vecteurs. En effet, si `n` et `m` sont deux entiers quelconques, `(tab n)` et `(tab m)` ne peuvent pas être comparés avec l'égalité telle qu'elle est définie dans *Coq*, puisque si `n` et `m` ne sont pas égaux, les types `(Tab n)` et `(Tab m)` sont deux types différents(ne sont pas convertibles)

Cela pose un problème si l'on veut par exemple exprimer que le seul vecteur de longueur O est le vecteur nul, autrement dit que si un vecteur est de longueur nulle c'est le vecteur vide.

On a commencé par énoncer le lemme suivant:

```
Lemma tab_nil_O1 : (s:Set)(n:nat)(t1:(Tab s n))
   (n=O) -> ( t1 = (Tnil s)).
```

ce qui n'est pas bien typé, en effet `t1` et `Tnil` sont de type différent.

Mais heureusement, il y a d'autres manières d'exprimer ce lemme.

En fait ce que nous avons définie est une famille de types indicés par les entiers; et on ne peut comparer que des objets de même indice.

L'égalité classique (dite "de Leibnitz") qui est définie inductivement par:

```
Inductive eq [A:Set;x:A] : A->Prop := refl_equal : (eq A x x)
```

ne convient pas pour travailler avec les type dépendants.

Nous avons donc définit une égalité dépendante pour le type `Tab`.

Elle est définie inductivement par:

```
Inductive eqn  :
 (n:nat)(x:(Tab n))(m:nat)(y:(Tab m))Prop :=
         eqn_Nil : (eqn O Tnil O Tnil)
        |eqn_Cons : (a,b:s)(n:nat)(x:(Tab n))(m:nat)(y:(Tab m))
              (a=b)->(eqn n x m y)
                ->(eqn (S n) (Tcons n a x) (S m) (Tcons m b y)).
```

Ensuite, on prouve un certain nombre de propriétés sur cette égalité

– que c'est une relation d'equivalence

```
Theorem eqn_refl :
(n:nat)(x:(Tab n))(eqn n x n x).

Theorem eqn_sym :
(n:nat)(x:(Tab n))(m:nat)(y:(Tab m))
 (eqn n x m y)->(eqn m y n x).

Theorem eqn_trans :
(n:nat)(x:(Tab n))(m:nat)(y:(Tab m))(p:nat)(z:(Tab p))
 (eqn n x m y)->(eqn m y p z)->(eqn n x p z).
```

– que si deux vecteurs sont égaux (au sens de `eqn`), toute propriété vérifiée par l'un l'est aussi par l'autre"...),

```
Theorem eqn_P :
       (n:nat)(x:(Tab n))(m:nat)(y:(Tab  m))
           (eqn n x m y)
                  ->(P:(n:nat)(Tab n)->Prop)(P n x)->(P m y).
```

Notons que le lemme que nous avons donné en exemple précédemment, c'est a dire que le seul vecteur de longeur zero est `Tnil` s'écrit maintenant

```
Theorem eqn_tab0_Tnil : (x:(Tab 0))(eqn 0 x 0 Tnil).
```

et que sa démonstration ne pose plus de problème.

Remarquons la notion d'égalité dépendante que nous avons définie dans le cas particulier des `Tab` est définie de façon beaucoup plus générale dans la bibliothèque Coq :(cf `THEORIE/STREAMS/Eqdep.v`)

- ```
  Variable U : Set.
  Variable P : U->Set.

  Inductive eq_dep [p:U;x:(P p)] : (q:U)(P q)->Prop :=
     eq_dep_intro : (eq_dep p x p x).
  ```

  Le lemme dont nous avons parlé plus haut s'écrit :

  ```
  Lemma tab_nil : (s:Set)(t1:(Tab s 0))
          (eq_dep nat  (Tab s) 0 t1 0 nil).
  ```

  c'est à dire qu'ici `U=nat`, le type de nos termes dépend d'un entier.

2.2.3 Les problème liés à l'élimination sur les type dépendant

Prenons d'abord un exemple: On veut définir la fonction `tail_tab` qui renvoie la queue d'un vecteur.

- Le type de `Tail_Tab` sera : `(n:nat)(x:(Tab n))(Tab (pred n))` c'est à dire que dans le type `Tab`, la longueur diminue, la fonction sera donc plus compliqué que dans le cas des listes ou `Tail` avait le type `list->list`, car elle est plus riche en information.

  Cela se traduit par un schéma d'élimination plus complexe
  ```

– comparons les schémas d'eliminations des listes et des vecteurs.
Le schéma d'élimination dépendant dans le cas des listes

```
Coq < Check list_rec.
list_rec
     : (s:Set)
        (P:(list s)->Set)
         (P (nil s))
          ->((s0:s)(l:(list s))(P l)->(P (cons s s0 l)))
             ->(l:(list s))(P l)
```

Dans celui des vecteurs

```
Coq < Check Tab_rec.
Tab_rec
 :(s:Set)
   (P:(n:nat)(Tab s n)->Set)
     (P O (Tnil s))
       ->((n:nat)
        (s0:s)(t:(Tab s n))(P n t)->(P (S n) (Tcons s n s0 t)))
          ->(n:nat)(t:(Tab s n))(P n t)
```

On constate que ce dernier est sensiblement plus compliqué mais surtout que le prédicat `P` a pour `(list s)->Set` dans le cas des listes alors qu'il est de type `(n:nat)(Tab s n)->Set` dans le cas des vecteurs.

Il faudra tenir compte de ce paramètre, (`n:nat`), supplémentaire lorsqu'on déclarera le type du terme d'élimination dans les `Case` et les `Match`.

Ainsi pour les listes, la fonction `tail` sécrivait:

```
Definition tail := [l:list ]<list >Case l of
           (* nil *)        nil
           (* cons s a m *) [a:s][m:list]m end
:list->list.
```

Pour les vecteur `tail_tab` s'écrit :

```
Definition tail_tab :=
 [n:nat][t:(Tab n)]<[p:nat](Tab (pred p))>Case t of
(* Tnil *)          Tnil
   (* Tcons s a m *)        [k:nat][a:s] [t1:(Tab k)] t1 end
 : (n:nat)(Tab n)->([p:nat](Tab (pred p)) n) .
```

Il y a donc un `[p:nat]` supplémentaire dans le type du terme d'élimination.

Il en sera de même pour toute les fonctions que nous écrirons sur les `Tab`

Remarque sur la conversion: On note que la réduction de `(pred p)` en un entier de type `nat` équivalent ne pose pas de problème, `pred (S n)` se réduit en `n` et `(pred O)` se réduit en `O`. Les choses ne sont pas toujours aussi simples. Ainsi dans le cas de la fonction `plus` qui est définie par récurrence sur son premier argument, le système convertira bien `(plus O n)` en `n` mais ne réduira pas `(plus n O)` en `n`.

2.2.4 Fonctions usuelles sur les vecteurs:

On donne maintenant les fonctions classiques de manipulation des vecteurs

La concaténation

```
Definition fun_apptab :=
[n1:nat]
[n2:nat]
[l1:(Tab n1)]
[l2:(Tab n2)]
(<[k:nat](Tab (plus k n2))>Match l1 with
   (* Tnil *) l2
 (* Tcons p a m *)[p:nat][a:s][m:(Tab p)][r:(Tab (plus p n2))]
                     (Tcons (plus p n2) a r)
 end).
```

On montre quelques propriétés de la concaténation des vecteurs:

– concaténer la liste vide ne change rien:

```
Lemma apptab_nil_end :
    (n:nat)(t:(Tab   n))(eqn  n  t  n (fun_apptab  0 n Tnil  t)).
```

– L'associativité:

```
Lemma ass_apptab :
 (n1,n2,n3:nat) (t1:(Tab n1))(t2:(Tab  n2))(t3:(Tab  n3))
 ( eqn
(plus n1 (plus n2 n3))
      (fun_apptab  n1 (plus n2 n3) t1 (fun_apptab   n2 n3 t2 t3))
(plus (plus n1 n2) n3)
      (fun_apptab  (plus n1 n2) n3 (fun_apptab  n1 n2 t1 t2) t3)).
```

Les fonctions `tail` et `Head`:

On a donné plus haut la fonction `tail_tab`; On donne ici une spécification un peut plus forte de cette notion:

```
Lemma Tl_Tab  : (n:nat)(t:(Tab n))
 {m:nat & {  t1:(Tab m) | (Ex [a: s] (eqn (S m) (Tcons m a t1) n t))
                       \/ ((Is_Tab_nil n t) /\ (Is_Tab_nil m t1))} }.
```

Et la fonction qui la réalise

```
Definition tail_tab2 :=
  [n:nat][t:(Tab n)]<[p:nat](nat*(Tab (pred p)))>Case t of
   <nat,(Tab 0)>(0,Tnil)
          [k:nat][a:s] [t1:(Tab k)] <nat,(Tab k)>(k,t1) end.

Realizer tail_tab2.
Program_all.
Left.
Exists a.
Apply eqn_refl.
Save.
```

C'est une variante de la fonction `tail_tab` que nous avons donné plus haut. Elle renvoie un couple dont la deuxième composante est la queue du vecteur en entrée tandis que la première composante est sa longueur.

La fonction `Hd_tab` se définie comme l'opérateur `Hd` que nous avions dans le cas des listes, il suffit de rajouter un `[p:nat]` dans le type du terme d'élimination; il vient:

```
Definition Hd_tab :=[n:nat][t1:(Tab n)]<[p:nat](Exc s)>Case t1 of
                           (error s)
   [k:nat][a:s][t2:(Tab k)] (value s a)
                            end.
```

C'est une preuve de la spécification suivante:

```
Lemma Hd_Tab2 :
  (n:nat)(t:(Tab n))
{a : s |
   Ex( [m:nat](Ex([t1:(Tab  m)](eqn (S m) (Tcons m a t1) n t ))))}
                    + {Is_Tab_nil n t}.

Realizer Hd_tab.
Program_all.
Exists k.
Exists t2.
Apply eqn_refl.
Save.
```

Les tests de nullité

On a un prédicat exprimant la nullité d'un vecteur:

```
Definition Is_Tab_nil := [n:nat][t:(Tab n)](eqn O Tnil n t).
```

Et on construit un programme permettant de décider si un vecteur est vide ou non:

```
Definition Tnil_dec :=
[n:nat][t:(Tab n)](<[k:nat]bool>Case t of
```

```
            (* Tnil *) true
         (* Tcons p a m *) [p:nat][a:s][m:(Tab p)]false
 end).
```

Cela permet de prouver le lemme suivant qui exprime que la relation "être nul" est décidable. C'est une preuve de la spécification suivante:

```
Lemma Null_Tab : (n:nat)(t:(Tab n)){(Is_Tab_nil n t)}
                                  +{~(Is_Tab_nil n t)}.
```

2.2.5 Particularité de `Realizer`

Signalons ici que `Realizer` travaille différemment selon qu'on lui donne en argument une fonction prédéfinie ou directement un lambda terme. En fait il travail au niveau des termes extraits (*i.e.* on n'est plus dans le C.I.C mais dans $\mathcal{F}_{\omega}^{idt}$); dans le premier cas, il réalise l'extraction avant de lier le lambda-terme au but à prouver, dans le second il faut lui donner directement le terme sous la forme extraite. Regardons cela sur un exemple:

Exemple

On donne le lemme suivant qui exprime que l'appartenance à un vecteur est décidable:

```
Lemma mem_tab :
 (a:s)(n:nat)(t:(Tab n))
      {(Intab a n t)}+{(AllS_tab  [b:s]~<s>b=a n t)}.
```

On prouvé cette spécification en utilisant les tactiques Coq usuelles.

Mais si l'on veut synthétiser cette preuve à partir de la fonction sensée la réaliser. On écrira

```
Realizer [a:s][n:nat][t:Tab]<bool>Match t with
         (* Tnil *) false
    (* Tcons p b m *) [p,b,m,H]
                  <bool>if (eq_dec a b) then true
                                        else H
                      end.
```

alors que la fonction définie indépendamment du `Realizer` s'écrie:

```
Fixpoint  mem_tab_dec [n:nat;t:(Tab n)] :(a:s)bool :=
     [a:s] <[p:nat]bool>Case t of
         (* nil *) false
     (* cons b m *) [m:nat][b:s][t1:(Tab m)]
                               <bool>Case  (eqs_dec a b) of
                                 [qq:?]true
                                 [ww:?](mem_tab_dec  m t1 a)
                                end
      end.
```

2.2.6 Autres fonctions utiles

La notion de projection

On aura deux prédicats, le premier spécifie qu'un élément `a` est la n-ième projection de la liste.

```
Inductive nth_spec_Tab :(n:nat)(Tab n)->nat->s->Prop :=
   nth_spec_Tab_O : (n:nat)(a:s)(t:(Tab n))
 (nth_spec_Tab (S n) (Tcons n a t) (S O) a)
   | nth_spec_Tab_S : (n:nat) (n1:nat)(a,b:s)(t:(Tab n))
 (nth_spec_Tab n t n1 a)->(nth_spec_Tab (S n) (Tcons n b t) (S n1) a).
```

Le second spécifie en plus que c'est la première apparition de `a`:

```
Inductive fst_nth_spec_Tab : (n:nat)(Tab n)->nat->s->Prop :=
  fst_nth_Tab_O : (n:nat)(a:s)(t:(Tab n))
       (fst_nth_spec_Tab (S n) (Tcons n a t) (S O) a)
| fst_nth_Tab_S : (n:nat)(n1:nat)(a,b:s)(t:(Tab n))
        (~a=b)->(fst_nth_spec_Tab  n t n1 a)->
        (fst_nth_spec_Tab (S n) (Tcons n b t) (S n1 ) a).
```

On prouve quelques lemmes sur ces notions:

– si `a` apparaît pour la première fois en n-ième position, alors il apparaît en n-ième position:

```
Lemma fst_nth_nth_Tab :
 (n:nat)(t:(Tab n))(n1:nat)(a:s)
   (fst_nth_spec_Tab n t n1 a)->(nth_spec_Tab n t n1 a).
```

– si n est l'indice d'un élément a alors n> O:

```
Lemma nth_lt_O_tab :
      (n:nat)(t:(Tab n))(n1:nat)(a:s)
         (nth_spec_Tab n t n1 a)->(lt O n1).
```

– si n est l'indice d'un élément a alors n est inférieure ou égale à la dimension du vecteur:

```
Lemma nth_le_long_Tab  :
            (n:nat)(t:(Tab n))(n1:nat)(a:s)
                      (nth_spec_Tab n t n1 a)->(le n1 n).
```

On donne pour terminer la fonction renvoyant la n-ième projection dans un Tab

```
Fixpoint Nth_func_Tab [n:nat;t:(Tab n)] : nat -> (Exc s)
  := [n1:nat]<[k:nat](Exc s)>Case t of
     (* nil *) (error s)
(* cons a m *)[q:nat][a:s][t':(Tab q)]
            <(Exc s)>Case n1 of
             (* O *) (error s)
             (* S m *) [m:nat]<(Exc s)>Case m of
                     (* O *) (value s a)
                     (* S p *) [p:nat](Nth_func_Tab q t' (S p))
 end
 end
end.
```

C'est une preuve de la spécification suivante :

```
Lemma Nth_Tab : (n:nat)(t:(Tab n))(n1:nat)
            {a:s|(nth_spec_Tab n t n1 a)}+{(n1=O)\/(lt n n1)}.
```

```
Realizer Nth_func_Tab.
Program_all.
Simpl; Elim n1; Auto.
(Elim o; Intro); [Absurd ((S p)=O); Auto | Auto].
Save.
```

Chapitre 3

Les structures abstraites

3.1 Les ensembles

Dans cette section, on cherche à définir une représentation des ensembles finis. L'idée est que les ensembles peuvent être vus comme des listes sans doublet, sur lesquelles l'égalité est définie modulo permutations. On dispose donc maintenant de tous les outils nécessaires à leur représentation.

3.1.1 Définition

On aimerait donc pouvoir définir les ensembles comme un sous-type des listes; mais il n'y a pas en Coq de moyen nécessaire pour représenter cette notion de sous-type. On va donc la simuler. Pour cela, on utilisera le mot clef `Inductive` bien que nos ensembles ne soient pas définis inductivement car, c'est la seule possibilité de Coq pour définir des *"type Somme"*. On a donc:

```
Inductive Ensemble : Set :=
    intro_ensemble : (l:(list s)) (sans_d s l)->Ensemble .
```

On peut remarquer que lors de l'extraction vers $\mathcal{F}_{\omega}^{idt}$on retrouve des listes:

```
Coq < Extraction Ensemble.
Ensemble ==> (list s)
```

On se donne d'autre part, une fonction de filtrage qui permet de retrouver la liste sous-jacente

```
Definition ext :=
        [l:Ensemble] <(list s)> Match l with [l1,p1:?] l1 end.
```

On aura aussi besoin d'une fonction permettant de récupérer une preuve que cette liste est sans doublet:

```
Definition extp := [e:Ensemble]
          <[e:Ensemble](sans_d s (ext e))> Case e of
              [l1:(list s)][p1:(sans_d s l1)]p1 end.
```

Notons au passage que la définition de cette fonction soulève de nouveau le problème du type des termes *d'élimination.*; On a ici une élimination dépendante. c'est à dire que le type du terme d'elimination dépend de l'objet que l'on élimine.

Légalité sur les ensembles, se déduit de celle des listes mais modulo la notion de permutation. Il vient donc:

```
Definition  eq_ens :=
   [ens1,ens2:Ensemble](Permut s (ext ens1) (ext ens2)).
```

En pratique, les notions que nous allons définir sur les ensembles découlent directement (via `ext`) de celles définies précédemment sur les listes. Quand il s'agira d'opérations, on pourra donc effectuer les opérations sur les listes sous-jacentes, puis prouver que le résultat est toujours une liste sans doublet pour pouvoir "remonter" ce résultat au niveau des ensembles grâce au constructeur `intro_ensemble`.

3.1.2 Quelques notions de base

L'appartenance à un ensemble s'écrit:

```
Definition In_ens_f := [a:s][e:Ensemble](fun_In s a (ext e)).
```

On n'aura parfois besoin du lemme suivant qui exprime que si un élément n'appartient pas à un ensemble, la liste formée en ajoutant cet élément en tête de la liste sous-jacente est toujours sans doublet.

```
Lemma cons_ens:
    (x:s)(e:Ensemble)
       (~(In_ens_f x e))->(sans_d s (cons s x (ext e))).
```

Cela permet essentiellement de définir la fonction `ajouter` qui prend un ensemble `e`, un élément `a` et une preuve que `a` n'est pas dans `e` et renvoie "l'union" de cet élément et de l'ensemble par :

```
Definition ajouter1 :=
   [e1:Ensemble][a:s][p:(~(In_ens_f a e1))]
     (intro_ensemble (cons s a (ext e1))(cons_ens a e1 p)).
```

On introduit aussi le cardinal d'un ensemble par :

```
Definition Card_ens := [e1:Ensemble](length s (ext e1)).
```

3.1.3 L'algèbre des ensembles

Les fonctions que nous avons vues jusqu'ici sur les ensembles avaient toutes un sens au niveau des listes. On va maintenant introduire des opérations "purements ensemblistes"; mais on les manipulera au niveau des listes. On va donc étre amener à définir sur les listes un certain nombres de fonctions utilitaires qui n'auront de sens que transposées sur le type des ensembles.

L'union

La définition classique de l'union de deux ensembles en mathématiques s'exprime par:

$$e_3 \; est \; l' \; union \; de \; e_1 \; et \; e_2 \; ssi \; (\forall x, ((x \in e_1) \vee (x \in e_2)) \leftrightarrow (x \in e_3))$$

la spécification de la fonction d'union sera donc :

```
Lemma existe_union_ens :
 (e1:Ensemble)(e2:Ensemble)
{l3:(list s)| (x:s)(In_Ensemble  x l1)\/(In_ensemble x l2)
                   <->(In_Ensemble x l3))}.
```

Mais comme nous l'avons expliqué on va effectuer les opération au niveau des listes puis on transposera le résultat au niveau des ensembles.

On définit donc le prédicat suivant

```
Definition est_l'union :=
```

```
      [l1,l2,l3:(list s)]
  (x:s)(((fun_In s x l1)\/(fun_In s x l2))
                      <->(fun_In s x l3)).
```

puis on donne la spécification :

```
Lemma existe_union :
 (l1,l2:(list s)){l3:(list s)|(est_l'union  s l1 l2 l3)}.
```

En pratique, il est facile d'écrire la fonction sensée réaliser l'union;

```
Definition union_c  := [l1,l2:(list s)]<(list s)> Match l1 with
          l2
          [x:s][l',l'':(list s)]<(list s)>Case (mem s x l2) of
                                          [x:?]l''
                                          [u:?](cons s x l'')
                                          end
           end.
```

Il reste à montrer que cette fonction réalise bien la spécification que l'on a donné. Pour cela, on utilise encore, `Realizer` qui lie le programme à sa spécification; puis la tactique `Program_all` qui tente de synthétiser la preuve. Cela nous génère quelques sous buts qu'il faut encore prouver.

On peut remarquer qu'un certain nombre des ces sous buts correspondent aux type des constructeurs d'une définition prédicative de la fonction d'union.

Mais pour l'instant, nous n'avons fait aucune hypothèse sur les listes de départ. On montre d'ailleurs que la concaténation vérifie aussi la spécification donnée ci dessus :

```
Lemma app_oui :
    (l1,l2:(list s))( est_l'union l1 l2 (fun-app s l1 l2)).
```

Mais, pour pouvoir transposer ces fonctions sur le type des ensembles, il faut, nous l'avons dit, prouver que si les données sont des listes sans doublets, alors le résultat est bien une liste sans doublet.

C'est clairement faux pour la concaténation. On va montrer que notre fonction `union_c`, vérifie bien cette condition;

c'est l'objet du lemme suivant qui se prouve par induction sur `l1`:

```
Lemma sans_d_union_c :
    (l1,l2:(list s))
  (sans_d s l1)->(sans_d s l2)->(sans_d s (union_c l1 l2)).
```

On peut donc à présent introduire l'union au niveau du type `Ensembles`; soit la fonction suivante:

```
Definition Union_ensemble :=
    [a,b:Ensemble] (intro_ensemble (union_c (ext a) (ext b))
        (sans_d_union_c (ext a) (ext b) (extp a) (extp b))).
```

C'est bien une preuve de la spécification de l'union des ensembles qui est donnée par le lemme suivant:

```
Lemma existe_union_ens :
 (e1:Ensemble)(e2:Ensemble)
{l3:(list s)| (x:s)(In_Ensemble  x l1)\/(In_ensemble x l2)
                       <->(In_Ensemble x l3))}.
```

On procède exactement de la même manière avec les autres opérations ensemblistes:

Pour la différence

On travaille d'abord au niveau des listes: on définit le prédicat "`est_la_diff`" par :

```
Definition est_la_diff:=
   [l1,l2,l3:(list s)]
    (x:s)(((Inf s x l1)/\(~(Inf s x l2)))<->(Inf s x l3)).
```

On donne une fonction sensée prouver cette spécification:

```
Fixpoint Differ_c [l:(list s)]:(list s)->(list s) :=
     [m:(list s)]<(list s) >Case l of
     (nil s)
     [a:s][l1:(list s) ]
```

```
                    <(list s)>Case (mem s a m) of
                      [qq:?](Differ_c  l1 m)
                      [ww:?](cons s a (Differ_c  l1 m))
                     end
      end.
```

et enfin on prouve via `Realizer, Program_all` et quelques lemmes intermediaires le lemme suivant :

```
Lemma existe_diff : (l1,l2:(list s))
        {l3:(list s)| (est_la_diff l1 l2 l3)}.
```

Et comme dans le cas de l'union, pour définir la différence sur les ensembles, il faut prouver que si l'on donne des listes sans doublet à notre fonction, le résultat est sans doublet[1].

```
Lemma sans_d_differ_c :
    (l1,l2:(list s))
   (sans_d s l1)->(sans_d s l2)->(sans_d s (differ_c l1 l2)).
```

On peut alors définir l'opérateur de différence sur les ensembles:

```
Definition Differ_ensemble :=
 [a,b:Ensemble] (intro_ensemble (Differ_c (ext a) (ext b))
  (sans_d_differ_c (ext a) (ext b) (extp a) (extp b))).
```

Qui est une preuve de la spécification de la difference des ensembles qui elle, est donnée par le lemme suivant:

```
Lemma existe_diff_ens :
 (e1:Ensemble)(e2:Ensemble)
{l3:(list s)| (x:s)(In_Ensemble  x l1)/\~(In_ensemble x l2)
                                  <->(In_Ensemble x l3))}.
```

[1]dans la pratique tout n'est pas encore prouvé et certains lemmes ont étés posés en axiomes

Pour l'intersection

Le principe est le même , On définit le prédicat "est_l'inter" par :

```
Definition est_l'inter:=
    [l1,l2,l3:(list s)](x:s)(((Inf s x l1)/\(Inf s x l2))
                                  <->(Inf s x l3)).
```

puis on a la fonction :

```
Fixpoint Inter_c [l:(list s)]:(list s)->(list s) :=
             [m:(list s)]<(list s) >Case l of
          (nil s)
          [a:s][l1:(list s) ]
                        <(list s)>Case  (mem s a m) of
                          [qq:?]( cons s a (Inter_b  l1 m))
                          [ww:?](Inter_b  l1 m)
                         end
             end.
```

et enfin, on prouve le lemme:

```
Lemma existe_inter :
   (l1,l2:(list s)){l3:(list s)| (est_l'inter l1 l2 l3)}.
```

Pour pouvoir écrire la fonction sur les ensembles on a besoin du lemme suivant:

```
Lemma sans_d_inter_c :
    (l1,l2:(list s))
(sans_d s l1)->(sans_d s l2)->(sans_d s (inter_c l1 l2)).
```

Il est alors possible de définir l'opérateur d'intersection des ensembles:

```
Definition Inter_ensemble :=
    [a,b:Ensemble] (intro_ensemble (Inter_c (ext a) (ext b))
        (sans_d_inter_c (ext a) (ext b) (extp a) (extp b))).
```

Qui est une preuve de la spécification de l'intersection des ensembles qui elle, est donnée par le lemme suivant:

```
Lemma existe_Inter
 (e1:Ensemble)(e2:Ensemble)
{e3:(Ensemble)| (x:s)(In_Ensemble  x e1)/\(In_ensemble x e2)
                                  <->(In_Ensemble x e3))}.
```

L'inclusion

On definit au niveau des listes le prédicat `incl`

```
Definition incl [l,m:list](a:s)(fun_In a l)->(fun_In a m).
```

Il prend deux listes en entrée et exprime le fait que tout élément qui appartient à la première appartient aussi à la seconde[2]; ce qui, au niveau des ensembles, nous permet d'exprimer l'inclusion d'un ensemble dans un autre par :

```
Definition Incl_ens := [e1,e2:Ensemble](incl s (ext e1) (ext e2)).
```

3.2 Les relations

On a maintenant les outils nécessaires à la représentation de nos relations. Elles sont, nous l'avons dit, représentées par des couples:

({liste des attributs} ,{ensemble de n-uplets})
Bien évidemment, le nombre d'attributs est égal à la longueur des n-uplets; on représentera donc, la "liste" d'attributs, et les tuples par des vecteurs, l'implémentation des ensembles étant celle que l'on a décrite plus haut, il vient pour nos relations, la représentation sous la forme d'un couple :

({vecteur},{Ensemble de vecteurs})

Attention : il faut noter ici que certaines opérations algébriques (comme la jointure) peuvent entrainer la formation de doublons qui devront donc être éliminés si l'on veut rester cohérent avec notre renprésentation des ensembles.

3.2.1 Définition

On définit donc deux domaines, `att` pour les attributs, `val`, pour les valeurs:

```
Variables att,val:Set.
```

[2]ce n'est pas l'inclusion multi-ensembliste car on ne tient pas compte du nombre d'occurences.

On redonne la défition des relations :

```
Definition Relation1 := [att:Set][n:nat][val:Set]
               (Tab att n)*(Ensemble (Tab val n)) .
```

On définit aussi une fonction de construction des relations, qui prend un vecteur et un ensemble de vecteurs et construit la relation, en posant :

```
Definition make_relation :=
[n:nat][att:Set][val:Set][t:(Tab att n)][l:(Ensemble (Tab val n))]
                <(Tab att n),(Ensemble (Tab val n))>(t,l).
```

3.2.2 Algèbre des relations

Pour pouvoir définir les opérations algébriques au niveau des relations il nous faut encore développer quelques fonctions utilitaires sur les listes et les ensembles de vecteurs:

Par abus de langage, on s'autorisera à employer pour ces fonctions le vocabulaire des operations de l'algèbre des bases de données[3].

Le produit cartésien

On aura des ensembles de vecteurs c'est à dire d'éléments qui sont eux même des éléments d'un produit cartésien d'ensemble (les domaines). Le produit cartésien étant associatif, l'ensemble résultant sera formé de la concaténation de chaque vecteur du premier ensemble avec tous les vecteurs du second.

On peut donc spécifier cela par:

$$\forall l_1, l_2, l_3, ((\forall x, (x \in l_1)) \wedge (\forall y, (y \in l_2))) \leftrightarrow ((x@y) \in l_3)$$

où " @ " représente la concaténation et pourvu que l_1, l_2, l_3 représentent bien des ensembles. (*i.e.* soit sans doublets).

On peut spécifier ça en Coq par:

```
Definition product :=
```

[3]pour des raisons de temps cetaines preuves de cette partie ont eté posées en axiomes afin de permettre d'exprimer les notions importantes

```
[n1,n2:nat]
 [l1:(list (Tab s n1))]
   [l2:(list (Tab s n2))]
     [l3:(list (Tab s (plus n1 n2)))]
       ((x:(Tab s n1))(y:(Tab s n2))
        ((Prop_In ? x l1)/\(Prop_In ? y l2))<->
    (Prop_In (Tab s (plus n1 n2)) (fun-apptab s n1 n2 x y) l3)).
```

Le lemme à prouver sera donc:

```
Lemma product :(n1,n2:nat)
(l1:(list (Tab s n1)))(l2:(list (Tab s n2)))
 {l:(list (Tab s (plus n1 n2))) | (product n1 n2 l1 l2 l)}.
```

On a défini les fonctions suivantes :
D'abord on a construit une fonction intermédiaire qui concatène un vecteur à tous les éléments d'une liste de vecteurs :

```
Fixpoint prod1_f
        [n:nat;m:nat;t1:(Tab s n);lt:(list (Tab s m))]
        : (list (Tab s (plus n m)))
        :=<(list (Tab s (plus n m)))>Case  lt of
           (* nil *) (nil (Tab s (plus n m)))
          (* cons *)[t2:(Tab s m)][l1:(list ( Tab s m))]
 (cons  (Tab s (plus n m)) (fun-apptab s  n m t1 t2) (prod1_f n m t1 l1) )
       end .
```

ensuite, on peut écrire la fonction qui fait le "produit" au sens défini plus haut. :

```
Fixpoint prod2_f
        [n:nat;m:nat;lt1:(list (Tab s m));lt2:(list (Tab s n))]
        : (list (Tab s (plus n m)))
        :=<(list (Tab s (plus n m)))>Case  lt2 of
 (* nil *) (nil (Tab s (plus n m)))
  (* cons *)[t2:(Tab s n)][l1:(list (Tab s n))]
   (fun-app (Tab s (plus n m))  (prod1_f n m t2 lt1) (prod2_f n m lt1 l1))
end.
```

Il faut prouver que notre fonction vérifie la spécification en démontrant le lemme suivant:

```
Lemma prod2_is_prod :
   (m,n:nat)(lt1:(list (Tab s m)))(lt2:(list (Tab s n)))
           (product m n lt1 lt2 (prod2_f m n lt2 lt1)).
```

De plus pour pouvoir définir cette notion au niveau des ensembles, il faut prouver que si les listes en entrée sont sans doublet, la liste resultante l'est aussi:

```
Lemma produit_sans_d :
 (m,n:nat)(lt1:(list (Tab s m)))(lt2:(list (Tab s n)))
  (sans_d s lt1)->(sans_d s lt2)->(sans_d (prod2_f m n lt2 lt1)).
```

On peut alors définir le produit de deux ensembles de vecteurs par la fonction :

```
Definition  produit_ensemble
 [n:nat;m:nat;et1:(Ensemble (Tab s m));et2:(Ensemble (Tab s n))] :=
    (intro_ensemble (prod2_f n m (ext (et1)) (ext et2))
      (produit_sans_d n m (ext (et1)) (ext et2))).
```

"La réstriction"

On peut la specifier par:

$$\forall e_1, e_3 \, \forall P, (\forall x, (x \in E_1); (P(x)) \leftrightarrow (x \in e_3))$$

c'est à dire que l'on veut garder tous les éléments d'une liste (puis d'un ensemble) qui verifie une propriété donnée.

On travaille d'abord au niveau des listes. On écrit donc :

```
Definition restriction :=
  [l1,l3:(list s)][P:(s->Prop)]
      ((x:s)(Prop_In s x l1)->(P x)->(Prop_In s x l3)).
```

Puis on prouvera le lemme:

```
Lemma restrict
 :(l1:(list s)){l:(list s) | (restriction l1 l R)}.
```

Une preuve de ce lemme sera donc une fonction qui prend une liste et un proprieté décidable et renvoie une liste de tous les éléments vérifiant cette propriété. (pour des raisons pratique, on ouvre une `section` , la propriété est exprimé comme une variable globale, et on se donne un axiome affirmant qu'elle est décidable on peut alors écrire la fonction de restiction.

```
section Restric.
Variable R:s->Prop
Hypothesis RS_dec : (a:s){(R a)}+{~(R a)}.

Definition restriction :=[l:(list s)]
          <(list s)>Match l with
           (* nil *) (nil s)
       (* cons a m *) [a:s][m:(list s)][H:(list s)]
              <(list s)>Case (RS_dec a)  of
                           [qq:?](cons s a H)
                           [ww:?]H
                        end
              end.
```

On peut vérifier grâce à `Realizer` et `Program_all` que cette fonction vérifie bien notre spécification.

Pour transposer tout cela au niveau des ensembles, il faut que les listes sans doublet soient closes par cette opération (ce qui est évident car on n'a rien rajouté) :

```
Lemma restriction_sans_d :
   (m,n:nat)(lt1:(list (Tab s m)))
        (sans_d s lt1)->(sans_d s lt2)->(sans_d (restriction lt1)).
```

On peut alors définir:

```
Definition  restriction_ensemble
    [n:nat;m:nat;et1:(Ensemble (Tab s m));et2:(Ensemble (Tab s n))]:=
(intro_ensemble (prod2_f n m (ext (et1)) (ext et2))
          (restriction_sans_d  (ext (et1))).
```

"La projection"

On peut la spécifier par:

$$\forall e_1, (\forall x, (x \in e_1)) \leftrightarrow (\Pi_v x) \in e_3)$$

Là encore, on commence par travailler au niveau des listes:

On a une fonction de projection sur les vecteurs, à savoir:

```
Fixpoint Nth_func_Tab [n:nat;t:(Tab n)] : nat -> (Exc s)
  := [n1:nat]<[k:nat](Exc s)>Case t of
     (* nil *) (error s)
(* cons a m *)[q:nat][a:s][t':(Tab q)]
              <(Exc s)>Case n1 of
                 (error s)
                 [m:nat]<(Exc s)>Case m of
                           (value s a)
                            [p:nat](Nth_func_Tab q t' (S p))
                          end
end
end.
```

Ceci nous a permis de définir la projection selon un axe (une colonne dans le cas des Bd), pour la définir selon un vecteur, il faudra appliquer cette fonction à tous les éléments de la liste; on définit donc une fonction `map` sur nos listes. On la redonne ici:

```
Variables s,s1:Set.

Definition map : (s->s1)->(list s)->(list s1) :=
   [f:(s->s1)][l:(list s)]<(list s1)>Match l with
          (nil s1)
          [a:s][l':(list s)][h:(list s1)](cons s1(f a) h)
              end.
```

3.2.3 Définition des opérations algébriques sur les relations

On a maintenant tout les outils nécessaires pour écrire les opérateurs de l'algèbre relationelle

3.2.4 Les fonctions sur les relations

L'opérateur de différence peut s'écrire:

```
Definition Minus_Relation:=
     [n:nat]
 [sh1:(Relatio att n val)][sh2:(Relatio att n val)]
(Make_relation
      ( Fst sh1) (Differ_Ensembles  (Tab val n) (Snd sh1) (Snd sh2)))

 : (n:nat)
        (Relatio att n val)
         ->(Relatio att n val)->(Tab att n)*(Ensemble (Tab val n)).
```

Pour l'union on a la fonction:

```
Definition Union_Relation:=
  [n:nat]
  [sh1:(Relatio att n val)][sh2:(Relatio att n val)]
```

Pour l'intersection on a:

```
Definition Inter_Relation:=
    [n:nat]
    [sh1:(Relatio att n val)][sh2:(Relatio att n val)]
 (Make_relation
   ( Fst sh1) (Inter_Ensemble (Tab val n) (Snd sh1) (Snd sh2)))
```

Pour le produit cartesien :

```
Definition Product_Relation:=
       [n,m:nat]
       [sh1:(Relatio att n val)][sh2:(Relatio att m val)]
 (make_relation
  (plus n m) att val  (fun-apptab att n m ( Fst sh1) (Fst sh2))
                   (prod2_f val  n m  (Snd sh2) (Snd sh1))).
```

La restriction s'écrit:

```
Definition Restiction2:=
    [n:nat]
       [sh1:(Relatio att n val][P:(Tab val n)->Prop]
          (Make_relation ( Fst sh1) (Restiction  (Snd sh1) P))
```

La projection sur une colonne s'écrit :

```
Definition Projection_Relation:=
    [n,m:nat]
    [sh1:(Relatio att n val][proj:(Tab nat n)]
(Make_relation
   (project ( Fst sh1) proj)  (project (Snd sh1) proj)).
```

La jointure peut s'écrire à partir de la restriction et du produit, elle est donnée par:

```
Definition Jointure:=
   [n,m:nat]
   [sh1:(Relatio att n val][sh2:(Relatio att m val][P:Prop]
    (Make_relation
      (Restriction2 (Produt2 n m sh1 sh2) P)).
end.
```

Chapitre 4

Critique et conclusions

4.1 Critique

On donne ici une altenative à notre représentation des vecteurs et on compare les deux implémentations.

Une autre implémentation des vecteurs Il existe une autre façon de définir le type des vecteurs: c'est de le voir comme une famille de "pseudo sous-types" du type des listes. On définirait donc inductivement une famille de type (indicée par N) en posant :

```
Inductive Vecteur [n:nat]: Set :=
  intro_vect : (l:(list s)) (eq nat (length s l) n)->(Vecteur n) .
```

De plus, on se donne une fonction de filtrage permettant de retrouver la liste sous-jacente:

```
Definition vect_list :=
 [n:nat][l:(Vecteur n)] <(list s)> Match l with [l1,p1:?] l1
end.
```

On a également besoin d'une fonction permettant de retrouver une preuve que cette liste est bien de longueur n:

```
Definition vect_leng :=
[n:nat][l:(Vecteur n)]
  <[ee:(Vecteur n)]((length s (vect_list n ee))=n)> Case l of
    [l1:(list s)][p1:((length s l1)=n)] p1 end.
```

Partant de cela on peut réutiliser toutes les fonctions définies sur les listes.Par exemple la concaténation:

```
Definition app_vet :=
  [n,m:nat]
   [v1:(Vecteur n)]
    [v2:(Vecteur m)]
(intro_vect
  (plus(length s (vect_list n v1)) (length s (vect_list m v2)))
       (fun-app s (vect_list n v1) (vect_list m v2))
         (length_app (vect_list n v1) (vect_list m v2))).
```

Elle utilise le lemme suivant:

```
Lemma length_app :(l1,l2:(list s))
     (length s (fun-app s l1 l2))=(plus (length s l1)

(length s l2)).
(Induction l1;Simpl;Auto).
Save.
```

qui prouve que la longueur de la liste résultant de la concaténation de listes est la somme de la longueur de ces listes.

On a utilisé le terme de "pseudo sous type" car ce que l'on a défini est en fait une famille de types `(Vecteur i)`, `i` appartenant à N. Chacun des types `(Vecteur i)` pouvant être "injecté" dans"le type des `list`". Remarquons que nos (`Tab n`) peuvent aussi être vu comme ça mais que "l'injection" est nettement moins visible.

Cette possibilité d'implémentation ne nous est apparue que tardivement et n'as pas encore pu être réellement comparée avec celle que nous avions jusque là. Mais les deux représentations paraissent à peu près équivalentes. Dans celle que nous avons choisie il faudra constament manipuler le filtrage sur les types dépendants; dans celle-ci, ce problème n'apparaît que dans la fonction `vect_leng` . Mais en contrepartie on doit constament prouver que les listes qui sont le résultat d'une opération vérifient le prédicat relatif à leur longueur.

Signalons encore une différence majeure au niveau de l'extraction et qui tend à nous faire préférer la deuxième représentation des vecteurs:

On redonne le terme extrait de `Tab`:

```
Coq < Extraction Tab.
Tab ==>
  Inductive Tab : Set := Tnil : Tab | Tcons : nat->s->Tab->Tab
```

On peut le représenter en *Caml* par

```
type 'a tab = tnil | tcons of (int*'a*'a tab);;
```

Et l'extraction de la fonction de concaténation nous donne:

```
Coq < Extraction apptab.
apptab2 ==>[n1,n2:nat]
    [l1,l2:Tab]
      (Fix F{F/2:nat->Tab->Tab=[_:nat]
                  ([t:Tab]
                    <Tab>Case t of
                         l2
                        ([n:nat]
                          [s0:s]
                           [t0:Tab]
                            (Tcons (plus n n2) s0
                                   (F n t0)))
                       end)
                   }
                n1 l1)
 :  nat->nat->Tab->Tab->Tab
```

Ce que l'on peut écrire en *Caml*

```
let  concat_tab n1 n2 t1 t2 = FF n1 t1
                where rec FF nn tt = match tt with
          tnil-> t2
         |(tcons ((n,a,r)))->(tcons ((n2+n), a, FF  n  r ));;
```

Pour le type `Vecteur`, l'extraction nous redonne les listes

```
Coq < Extraction Vecteur.
Vecteur ==> (list s)
```

et l'extraction de la fonction de concaténation donne:

```
Coq < Extraction app_vet.
app_vet ==>[n,m:nat]
    [v1,v2:(list s)](fun_app s (vect_list n v1) (vect_list m v2))
 :  nat->nat->(list s)->(list s)->(list s)
```

On peut se convaincre que les deux entiers en argument ne jouent plus aucun rôle, et pourraient donc facilement être abandonné, en extrayant `vect_list`

```
Coq < Extraction vect_list.
vect_list ==>[_:nat] ([l:(list s)]l)
 :  nat->(list s)->(list s)
```

En abandonnant ces entiers on retrouve la fonction de concaténation des listes, soit *Caml*:

```
let rec concat l1 l2 = match l1 with
                  []->l2
                  |x::r->x::(concat r l2);;
```

4.2 Conclusion

On a donc donné une implémentation des listes, au dessus de laquelle on a pu construire une représentation certifiée des ensembles. Nos ensembles étant vus comme des sous types des listes. L'idée a été de developper tous les outils au niveau des listes puis de les transposer ensuite sur les ensembles. Puis on a introduit une représentation des vecteurs en termes de listes de longueur donnée . On a alors pu représenter les relations, et écrire les fonctions de manipulation de l'algèbre relationnelle, mais tout n'a pas été specifié et prouvé.

La suite logique de ce travail sera donc de terminer la spécification des opérations algèbriques sur les relations et, à plus long terme, de s'intéresser aux problèmes de contraintes d'intégrité, afin d'aller vers une représentation certifiée d'un système de gestion des bases de données relationnelles.

Bibliographie

[Barendregt92] H. Barendregt. Lambda calculi with types. In *Handbook of Logig in Computer Science*, volume 2. Oxford science publications, Oxford, 1992.

[Dowek91] G. Dowek. *Démonstation Automatique dans le Calcul des Constructions.* These de Doctorat, Université Paris 7, 1991.

[Gardarin95] G. Gardarin. *Bases de données, les systémes et leurs langages.* Eyrolles, Paris, 1995.

[JGirard89] P. Taylor J.Y. Girard, Y. Lafont. *Proofs and Types.* Canbridge Tracts in Theoretical Computer Science, 1989.

[Lalement90] R. Lalement. *Logique, Réduction, Résolution.* Masson, 1990.

[MBouzeghoub90] P. Pucheral M. Bouzeghoub, M. Jouve. *Systèmes de Bases de Données, des techniques d'implantation à la conception de schémas.* Eyrolles, Paris, 1990.

[Parent95] C. Parent. *Synthèse de preuves de programmes dans le calcul des constructions inductives.* These de Doctorat, Ecole Normale Supérieure de Lyon, 1995.

[Paulin-Mohring89] C. Paulin-Mohring. *Extraction de programmes dans le calcul des construction.* These de Doctorat, Université Paris 7, 1989.

[Ullman80] J.D. Ullman. *Principles of Database Systems.* Computeur Science Press, 1980.

[Werner94] B. Werner. *Une Théorie des Construction Inductives.* These de Doctorat, Université Paris 7, 1994.