Preuves Interactives et Applications

Christine Paulin & Burkhart Wolff
http://www.lri.fr/~paulin/PreuvesInteractives

Université Paris-Saclay

Master Informatique FIIL, 2016–17

Introduction

Ce que l'on a dans la logique

- λ-calcul simplement typé : constantes, variables et fonctions
- quantification d'ordre supérieur sur les fonctions et les prédicats

Ce que l'on veut représenter

- des types de données structurés : listes, arbres...
- des fonctions (programmes) opérant sur ces structures
- de nouveaux concepts correspondant à des descriptions inductives de relations (systèmes de transition, langages...)

Les limites de la logique du premier ordre

- Introduire de nouvelles notions
  - « une bibliothèque pour les piles »
  - « G est un graphe »
- Vision axiomatique : de nouveaux symboles de prédicats et axiomes
  - garantir que les axiomes sont cohérents (sinon on peut tout prouver)
  - garantir que les axiomes sont suffisants pour prouver toutes les propriétés qui nous intéressent
- Limitations :
  - on ne peut pas décider si une théorie est cohérente
  - certaines théories ne sont pas axiomatisables au premier ordre (théorie des groupes finis)
- Vision définitionelle dans des théories fixées
  - fonctions *primitives récursives* en arithmétique
  - constructions *ensemblistes* en théorie des ensembles
  - construction « *fonctionnelles* » en logique d'ordre supérieur

Exemple de la pile

```
Variable P A : Type.
Variable empty : P.
Variable push : A → P → P.
Variable top : P → A.
Variable pop : P → P.
Variable length : P → nat.
(* Hypothesis *)
Hypothesis le : length empty = 0.
Hypothesis lp: ∀ a p, length (push a p) = length p + 1.
Hypothesis tp : ∀ a p, top (push a p) = a.
Hypothesis pp: ∀ a p, pop (push a p) = p.
(* Properties *)
Goal ∀ a1 a2 a3,
     top (pop (push a1 (push a2 (push a3 empty)))) = a2.
```

La théorie peut devenir incohérente

```
Variable a : A.
Variable p : P.
Hypothesis defp : p = push a p.
Goal False.
```

Exemple des graphes

La logique du premier ordre manque parfois d'expressivité

```
Variable V E : Type.
Variable adj : E → A → bool.

(* au moins un sommet adjacent à chaque arète *)
Hypothesis adj1 : ∀ e, ∃ v, adj e v = true.
(* au plus deux sommets adjacents à chaque arète *)
Hypothesis adj2 : ∀ e v1 v2 v3,
   adj e v1 = true → adj e v2 = true → adj e v3 = true
    → v1=v2 \/ v1=v3 \/ v2=v3.
```

- On ne peut pas axiomatiser le fait que le graphe est fini (sans fixer la taille)
- Comment raisonner sur des chemins (finis ou infinis) ?

Cours 3

Relation inductive

- Une manière naturelle de *définir* une relation par ses *propriétés* :
  - n'est possible que pour certaines formes de propriétés
  - caractérisation : la plus petite relation qui satisfait les propriétés
  - s'appuie sur un théorème de point fixe (Tarski) : $F(R) = R$
- Se définit aisément en logique d'ordre supérieur
- Applications
  - description des systèmes logiques
  - description de la sémantique des langages de programmation
  - fermeture transitive, chemins
  - définition récursive de fonctions

Exemples

- Ensemble $\texttt{FP}$ des piles finies : plus petit ensemble qui contient $\texttt{empty}$ et qui est clos par l'opération $\texttt{push}$
  - $\texttt{FP}\,\texttt{empty}$
  - $\forall a\,p, \texttt{FP}\,p \Rightarrow \texttt{FP}\,(\texttt{push}\,a\,p)$
  - Règles d'inférence :

$$\frac{}{\texttt{FP}\,\texttt{empty}} \qquad \frac{\texttt{FP}\,p}{\texttt{FP}\,(\texttt{push}\,a\,p)}$$

- Relation L entre les piles et les entiers : plus petite relation telle que
  - $L\,\texttt{empty}\,0$
  - $\forall n\,a\,p, L\,p\,n \Rightarrow L\,(\texttt{push}\,a\,p)\,(1+n)$
  - Règles d'inférence :

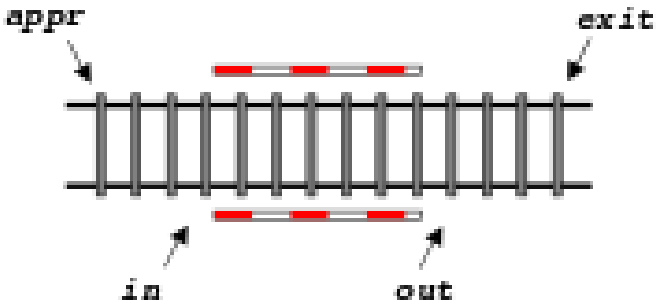
$$\frac{}{L\,\texttt{empty}\,0} \qquad \frac{L\,p\,n}{L\,(\texttt{push}\,a\,p)\,(1+n)}$$

Les piles finies sont celles qui ont une longueur (domaine de la relation)

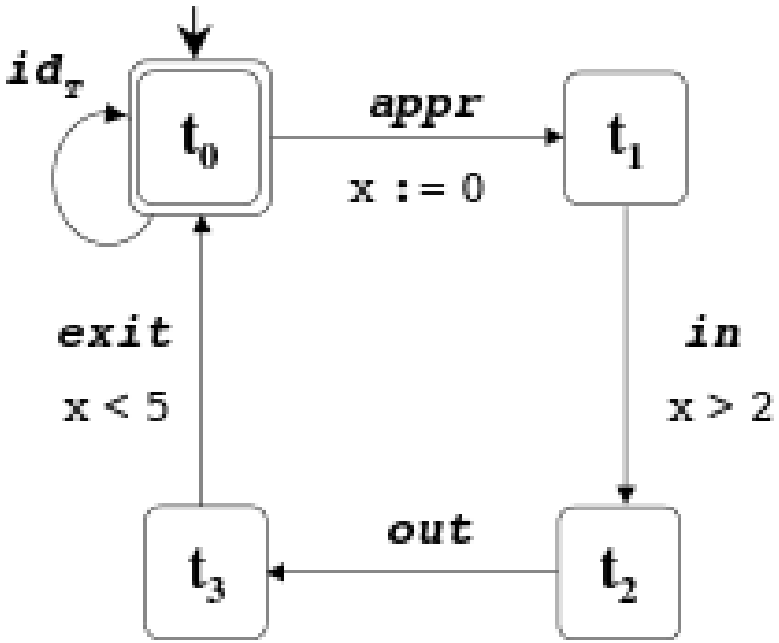
$$\texttt{FP}\,p \Leftrightarrow \exists n, L\,p\,n$$

Systèmes de transitions

Automate temporisé : on ajoute des horloges, et des contraintes de temps.



- Le train signale son approche **au moins 2 mn** avant d'entrer dans la section gardée ;
- Il ne s'écoule **pas plus de 5 mn** entre l'approche et la sortie de zone contrôlée



(source `www-master.ufr-info-p6.jussieu.fr/2005/IMG/pdf/Automates_tempo.pdf`)

Modélisation du train

- L'état du système comprend la valeur de l'horloge x
- Les actions intègrent le passage du temps δ
- Relation $(s, x) \longrightarrow^{a|\delta} (s', x')$

Propriétés des transitions élémentaires

- $\forall x\, \delta, (t_i, x) \longrightarrow^{\delta} (t_i, x + \delta)$
- $\forall x, (t_0, x) \longrightarrow^{\mathtt{appr}} (t_1, 0)$
- $\forall x, x > 2 \Rightarrow (t_1, x) \longrightarrow^{\mathtt{in}} (t_2, x)$
- $\forall x, (t_2, x) \longrightarrow^{\mathtt{out}} (t_3, x)$
- $\forall x, x < 5 \Rightarrow (t_3, x) \longrightarrow^{\mathtt{exit}} (t_0, x)$

Traces d'exécution

$$(s_0, x_0) \longrightarrow^{(a_1,\tau_1)} (s_1, x_1) \longrightarrow \cdots \longrightarrow^{(a_n,\tau_n)} (s_n, x_n)$$

Reconnaissance du mot $a_1, \ldots, a_n$ après un temps τ

- état initial : $\dfrac{}{\mathtt{valid}(t_0, 0, \tau, \epsilon)}$
- étape : $\dfrac{\mathtt{valid}(t_i, x, \tau, m) \quad [(t_i, y) \longrightarrow^{a} (t_j, y')] \quad x < y}{\mathtt{valid}(t_j, y', \tau + (y - x), ma)}$

Sémantique des langages

- Expressions entières : e
- Commandes : $x = e | c_1 ; c_2 | \textbf{while}(e)\{c\} | \textbf{if}(e)\{c\}$
- Mémoire $s \in M$

Sémantique

- évaluation des expressions : $\texttt{eval}(e, s) \in \mathbb{N}$
- mémoire : fonction des variables vers les entiers
- sémantique naturelle : $[(s, c) \longrightarrow s']$
  - $[(s, x = E) \longrightarrow (s + \{x \mapsto \texttt{eval}(e, s)\})]$
  - $[(s, c_1) \longrightarrow s_1] \Rightarrow [(s_1, c_2) \longrightarrow s_2] \Rightarrow [(s, c_1 ; c_2) \longrightarrow s_2]$
  - $\texttt{eval}(e, s) = 0 \Rightarrow [(s, \textbf{if}(e)\{c\}) \longrightarrow s]$
  - $\texttt{eval}(e, s) \neq 0 \Rightarrow [(s, c) \longrightarrow s_1] \Rightarrow [(s, \textbf{if}(e)\{c\}) \longrightarrow s_1]$
  - $\texttt{eval}(e, s) = 0 \Rightarrow [(s, \textbf{while}(e)\{c\}) \longrightarrow s]$
  - $\texttt{eval}(e, s) \neq 0 \Rightarrow [(s, c) \longrightarrow s_1] \Rightarrow [(s_1, \textbf{while}(e)\{c\}) \longrightarrow s_2]$
    $\Rightarrow [(s, \textbf{while}(e)\{c\}) \longrightarrow s_2]$

Théorème de point fixe de Tarski

Définition

Un ensemble ordonné $(A, \leq)$ est un treillis si deux éléments x et y ont une borne supérieure (notée $x \sqcup y$) et une borne inférieure (notée $x \sqcap y$).
Le treillis $(A, \leq)$ est complet si tout ensemble X (même infini) admet une borne supérieure (notée $\bigsqcup X$).

$$\bigsqcap X = \bigsqcup \{m \in A | \forall x \in X, m \leq x\} \quad \top = \bigsqcup A$$

Theorem (Théorème de Tarski : point fixe de fonction monotone)

Soit A un treillis complet et f une application croissante de A dans A, alors l'ensemble des point-fixes de f (ie $\{x \in A \mid f(x) = x\}$) est non-vide et admet un plus petit (et un plus grand) élément.

Preuve du théorème de Tarski

Ensemble des pre-point-fixes $F \stackrel{\text{def}}{=} \{x \in A \mid f(x) \leq x\}$, $a \stackrel{\text{def}}{=} \bigsqcap F$.

1. $\forall x, f(x) \leq x \Rightarrow a \leq x$.
2. $\forall y\, (\forall x, f(x) \leq x \Rightarrow y \leq x) \Rightarrow y \leq a$.

On montre

- $f(a) \leq a$ par (2) :
  Soit x tq $f(x) \leq x$, par (1) on a $a \leq x$ et donc comme f est croissante $f(a) \leq f(x)$ et par transitivité $f(a) \leq x$ et donc par (2) $f(a) \leq a$. Donc $a \in F$.
- $a \leq f(a)$: on utilise (1) avec $x = f(a)$. il suffit de montrer $f(f(a)) \leq f(a)$ qui est une conséquence de la monotonie et de la propriété $f(a) \leq a$ montrée précédemment.
- Donc a est un point fixe, c'est le plus petit car si on a un autre point fixe x, il appartient à F et donc il est plus grand que a.

Remarques

- $(\wp(A), \subseteq)$ est un treillis complet $\bigsqcup X = \bigcup X \qquad \bigsqcap X = \bigcap X$
- le plus grand point fixe se construit de manière analogue comme la borne supérieure de l'ensemble des post-point fixes : $G \stackrel{\text{def}}{=} \{x \in A \mid x \leq f(x)\}$
- la condition de monotonie est essentielle $(\wp(\mathbb{N}), \subseteq)$
  - $\forall x, x \notin M \Rightarrow (x+1) \in M$
  - $x \in M \Leftrightarrow \exists y, x = y + 1 \wedge y \notin M$
  - Solution $M \stackrel{\text{def}}{=} \bigcap\{X | \{x | \exists y, x = y + 1 \wedge y \notin M\} \subseteq X\}$
  - On a $M \subseteq \texttt{pair}$ et $M \subseteq \texttt{impair}$ donc $M = \emptyset$ qui contredit $\forall x, x \notin M \Rightarrow (x+1) \in M$
- Codage en logique d'ordre supérieur

$$M\,x \stackrel{\text{def}}{=} \forall X : A \to \textbf{prop}, (\forall \vec{y}, f(X)\,y \Rightarrow X\,y) \Longrightarrow X\,x$$

Relation inductive en logique d'ordre supérieur

- On introduit un nouveau nom de relation M avec une arité
  $\alpha_1 \to \cdots \alpha_p \to \mathbf{prop}$
- On spécifie un nombre fini d'opérations de *construction* pour la relation
  - $\forall y_1 \ldots y_l, G_1(M) \Rightarrow \cdots \Rightarrow G_q(M) \Rightarrow M\, u_1 \ldots u_p$

  $G_k(M)$ doit être monotone en M
- Le système garantit
  - $M : \alpha_1 \to \cdots \alpha_p \to \mathbf{prop}$
  - Les propriétés de construction sont vérifiées pour M
  - Principe d'induction associé (minimalité)

$$\begin{array}{l}\forall X : \alpha_1 \to \cdots \alpha_p \to \mathbf{Prop},\\ C_1(X) \Rightarrow \ldots \Rightarrow C_n(X)\\ \Rightarrow \forall z_1 \ldots z_p, M\, z_1 \ldots z_p \Rightarrow X\, z_1 \ldots z_p\end{array}$$

  - Principe d'inversion dérivé

$$\begin{array}{l}\forall z_1 \ldots z_p, M\, z_1 \ldots z_p \Rightarrow\\ (\ldots \vee (\exists y_1 \ldots y_l, G_1(M) \wedge \cdots \wedge G_q(M) \Rightarrow z_1 = u_1 \wedge \ldots \wedge z_p = u_p) \vee \ldots)\end{array}$$

Exemple en Coq

```
Variable P A : Type.
Variable empty : P.
Variable push : A → P → P.

Inductive FP : P → Prop :=
   FPe : FP empty
 | Fpp : ∀ a p, FP p → FP (push a p).

Inductive L : P → nat → Prop :=
 | Le : L empty O
 | Lp : ∀ a p n, L p n → L (push a p) (1+n).

Lemma FPL : ∀ p, FP p ↔ ∃ n, L p n.
Lemma LPinv0 : ∀ p n, L p 0 → p = empty.
Lemma LPinvn0 : ∀ p n, L p (1+n) → p <> empty.
```

Exemple en Isabelle

```
typedecl P
typedecl A

consts empty :: P
consts push :: "A ⇒P ⇒P"

inductive FP :: "P ⇒bool"
 where
   FPe : "FP empty"
 | Fpp : "FP p ⟹FP (push a p)"

inductive L :: "P ⇒nat ⇒bool"
 where
   Lem : "L empty 0"
 | Lpu : "L p n ⟹L (push a p) (1+n)"
```

Cours 3

Types de données

- En informatique, on travaille souvent avec des données structurées
  - bloc : étiquette symbolique + taille (arité)
  - listes : `nil` (0), `cons` (2)
  - arbres binaires : `leaf` (0), `node` (3)
  - record : un seul constructeur d'arité fixée et des fonctions de projection
- Opérations de base
  - créer un bloc en se donnant l'étiquette et les éléments pour initialiser les champs
  - raisonner par cas sur l'étiquette des blocs
  - accéder aux composantes d'un bloc
- Notion mathématique associée : structure de termes (algèbre initiale) associé à une signature

Exemples en Coq

```
Inductive nat : Type :=  O : nat | S : nat → nat.
Inductive list (A : Type) : Type :=
    nil : list A | cons : A → list A → list A.
Inductive tree (A : Type) : Type :=
    leaf : tree A | node : A → tree A → tree A → tree A.
Inductive itree (A B : Type) : Type :=
    ileaf : itree A B | inode : A → (B → itree A B) → itree A B.
```

Traitement en logique d'ordre supérieur

- On introduit un nouveau nom de type T
- On spécifie le type d'un nombre fini d'opérations de *construction* pour le type T
  - $G_1(T) \to \cdots \to G_q(T) \to T$ ($G_k(T)$ monotone en T)
- Le système garantit
  - $T \in$ **type**
  - Les opérations de construction sont des constantes $c_1, \ldots, c_n$
  - Les constructeurs sont distincts $(c_i\ t_1 \ldots t_k) \neq (c_j\ u_1 \ldots u_l) \quad i \neq j$
  - Les constructeurs sont injectifs

$$(c_i\ t_1 \ldots t_k) = (c_i\ u_1 \ldots u_k) \Rightarrow t_1 = u_1 \wedge \ldots t_k = u_k$$

  - Principe de définition par filtrage (complet)

```
match x with
| c1 y1..yl ⇒ t1
...
| ci x1..xk ⇒ ti
...
end
```

Récurrence et appel récursif

L'ensemble T est spécifié comme étant le plus petit ensemble clos par les opérations de construction

- Récurrence structurelle :

$$\begin{array}{l} \forall X : T \to \textbf{Prop}, \\ C_1(X) \Rightarrow \ldots \Rightarrow C_n(X) \\ \Rightarrow \forall z : T, X\, z \end{array}$$

$C_i(X)$ se construit en fonction de la forme du type du constructeur

$$C_i(T) \stackrel{\text{def}}{=} G_1(T) \to \cdots \to G_q(T) \to T \qquad G_j(T) = T$$

$$C_i(X) \stackrel{\text{def}}{=} \forall (x_1 : G_1(T)) \cdots (x_k : G_q(T)), \ldots X\, x_j \ldots \Rightarrow X\,(c_i\, x_1 \ldots x_k)$$

- Définition par point-fixe structurel

$$f\,(c_i\, x_1 \ldots x_k) = \ldots (f\, x_j) \ldots$$

Définition récursive par cas : exemples

```
Fixpoint natR f g n : X := match n with
   O ⇒ f | S p ⇒ g p (natR f g p)
end.
(* natR : X → (nat → X → X) → nat → X *)

Fixpoint listR f g (l:list A) : X := match l with
   nil ⇒ f | cons a p ⇒ g a p (listR f g p)
end.
(* listR : X → (A → list A → X → X) → list A → X *)

Fixpoint treeR f g (t:tree A) : X := match t with
   leaf ⇒ f | node a l r ⇒ g a l r (treeR f g l) (treeR f g r)
end.
(* treeR : X → (A → tree A → tree A → X → X → X)
          → tree A → X *)
Fixpoint itreeR f g (t: itree A B) : X := match t with
   ileaf ⇒ f | inode a F ⇒ g a F (fun x ⇒ itreeR f g (F x))
end.
(* itreeR : X → (A → (B → itree A B) → (B → X) → X)
           → itree A B → X *)
```

Principe induction : exemples

```
nat_ind : ∀ P : nat → Prop,
       P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
list_ind : ∀ (A : Type) (P : list A → Prop),
       P nil →
       (∀ (a : A) (l : list A), P l → P (a :: l)%list) →
       ∀ l : list A, P l
tree_ind : ∀ (A : Type) (P : tree A → Prop),
       P (leaf A) →
       (∀ (a : A) (t : tree A),
        P t → ∀ t0 : tree A, P t0 → P (node A a t t0)) →
       ∀ t : tree A, P t
itree_ind : ∀ (A B : Type) (P : itree A B → Prop),
       P (ileaf A B) →
       (∀ (a : A) (i : B → itree A B),
        (∀ b : B, P (i b)) → P (inode A B a i)) →
       ∀ i : itree A B, P i
```

Exemples en Isabelle

```
datatype nat = Z | S nat
datatype 'a list = nil | cons 'a "'a list"
datatype 'a tree = leaf | node 'a "'a tree" "'a tree"
datatype ('a,'b) itree =
   ileaf | inode 'a "'b => ('a,'b) itree"

(*
TypesInductifs.nat.induct: ?P Z ⟹(⋀x. ?P x ⟹?P (S x)) ⟹?P ?nat
TypesInductifs.list.induct:
   ?P nil ⟹(⋀x1 x2. ?P x2 ⟹?P (cons x1 x2)) ⟹?P ?list
TypesInductifs.tree.induct:
   ?P leaf ⟹(⋀x1 x2 x3. ?P x2 ⟹?P x3 ⟹?P (node x1 x2 x3)) ⟹?P ?tre
TypesInductifs.itree.induct:
   ?P ileaf ⟹
   (⋀x1 x2.
      (⋀x2a. x2a ∈range x2 ⟹?P x2a) ⟹?P (inode x1 x2 x3)) ⟹
   ?P ?itree
*)
```

Exemples en Isabelle (2)

```
fun natR (* :: "'x ⇒(nat ⇒'x ⇒'x) ⇒nat ⇒'x" *)
where
   "natR f g Z = f"
 | "natR f g (S p) = g p (natR f g p)"

fun listR where
   "listR f g nil = f"
 | "listR f g (cons a p) = g a p (listR f g p)"

fun treeR where
   "treeR f g leaf = f"
 | "treeR f g (node a l r) = g a l r (treeR f g l) (treeR f g r)"
```

Terminaison

- on peut évaluer les fonctions définies récursivement (**Eval compute in/value**)
- condition de positivité : terminaison et cohérence

```
type l = Lam of (l → l)
(* Lam : (l → l) → l *)
let _ = fun f → Lam f
let app t u = match t with (Lam f) → f u
(* app : l → l → l *)
let delta : l = Lam (fun x → app x x)
let bigdelta : l = app delta delta
(* boucle *)
```

- Valeur (terme clos en forme normale) : $c_i\ t_1 \ldots t_p$ avec c_i un constructeur
- Condition syntaxique de décroissance structurelle dans les appels récursifs

Types vides

- Des spécifications récursives peuvent être vides

```
Inductive E : Type := C : E → E.
(* E_ind : ∀ P : E → Prop,
        (∀ e : E, P e → P (C e)) → ∀ e : E, P e *)
Lemma Eempty : ∀ x:E, False.
```

- Dans Isabelle/HOL, les types sont non vides, la définition précédente est refusée

Définition de types non récursifs

Le même mécanisme est utilisé pour définir des types non récursifs : types finis, records.

```
Inductive couleur : Type := Pique | Coeur | Carreau | Trefle.
Inductive figure : Type := As | Roi | Dame | Valet | dix | neuf |
Record carte : Type := mkc {coul : couleur; fig : figure}.
(* carte_ind : ∀ P : carte → Prop,
       (∀ (coul : couleur) (fig : figure),
        P {| coul := coul; fig := fig |}) → ∀ c : carte, P c
mkc  : couleur → figure → carte
coul : carte → couleur
fig  : carte → figure
*)
```

En Isabelle/HOL, les records ont de plus une possibilité d'extension

```
datatype couleur = Pique | Coeur | Carreau | Trefle
datatype figure = As | Roi | Dame | Valet | dix | neuf | huit | sept
record carte = coul :: couleur fig :: figure
```

Tactiques pour les inductifs en Coq et Isabelle

- Tactiques Coq
  - `induction` n (terme)
  - `destruct` t analyse par cas d'un terme pour prouver $G(t)$, prouve $\forall x : T, G(x)$
  - `inversion` n inversion d'une instance d'une définition inductive

$$\frac{\forall x, M(x) \Rightarrow x = t \Rightarrow G}{M\,t \Rightarrow G(t)}$$

  - `simpl` f permet de simplifier une définition récursive
- Tactiques Isabelle
  - **apply**(`erule` <*name-elimination-rule*>)
    typiquement *impE*, *notE*, *allE*, ...,
    mais aussi *name*.cases ou *name*.induct
  - **apply**(`subst` <*simplification-rule*>) ou pour les hypothèses :
    apply(`subst` (`asm`) <*simplification-rule*>)
  - **apply**(`simp`) ou **apply**(`simp` add : <*simplification-rule*>) :
    récriture exhaustive.

Théorie des définitions inductives en Coq et Isabelle

- Coq :
  - primitives dans le langage,
  - mécanisme uniforme pour les types inductifs et les relations inductive
  - tout terme clos dans un type inductif se calcule
- Isabelle/HOL
  - codage à l'ordre supérieur à partir des constructions de base
  - mécanisme ad-hoc pour le calcul

Cours 3

Preuve de programmes

- Définir le type `list` des listes polymorphes avec deux constructeurs `nil` et `cons`
- Définir récursivement la fonction `app` de concaténation de deux listes
- Définir récursivement la fonction `map` qui applique une fonction f à tous les éléments d'une liste
- Montrer que $\texttt{map}\, f\, (\texttt{app}\, l\, l') = \texttt{app}\, (\texttt{map}\, f\, l)\, (\texttt{map}\, f\, l')$
- Définir récursivement la fonction `filter` qui extrait d'une liste la liste des éléments qui vérifient une condition p
- A quelle condition sur f, p et q a-t-on $\texttt{map}\, f\, (\texttt{filter}\, p\, l) = \texttt{filter}\, q\, (\texttt{map}\, f\, l)$?
  Faire la preuve correspondante

Relation d'équivalence

- Définir inductivement une relation `equiv` qui dit que deux listes sont une permutation l'une de l'autre
- Montrer que c'est une relation d'équivalence
- Montrer que si $\texttt{equiv}\, l\, (\texttt{app}\, m_1\, m_2)$ alors $\texttt{equiv}\, (\texttt{cons}\, a\, l)\, (\texttt{app}\, m_1\, (\texttt{cons}\, a\, m_2))$

Théorie incohérente

On se donne la théorie suivante pour représenter des tableaux de bits.

```
Variables T : Type.
Variable length : T → nat. (* length of arrays *)
Variable get : T → nat → bool. (* access *)
Variable update : T → nat → bool → T. (* update *)
Variable make : nat → bool → T. (* initialisation *)
(* Axioms *)
Hypothesis get_make : ∀ n k b, get (make n b) k = b.
Hypothesis length_make : ∀ n b, length (make n b) = n.
Hypothesis get_upd_eq : ∀ t n b,
           n < length t → get (update t n b) n = b.
Hypothesis get_upd_diff : ∀ t n m b, n < length t → m <> n
          → get (update t n b) m = get t m.
Hypothesis length_upd : ∀ t n b, length (update t n b) = length t.
Hypothesis ext : ∀ t1 t2, length t1 = length t2
          → (∀ n, n < length t1 → get t1 n = get t2 n)
          → t1 = t2.
```

1. montrer qu'on peut dériver une preuve de faux dans ce contexte ;
2. que proposez-vous pour corriger ce problème ?
3. comment s'assurer que la nouvelle théorie est cohérente ?