Cours 2-7 Preuves Constructives
Examen terminal
Mardi 15 février 2005

L'examen se compose de trois parties. La solution de chaque problème doit être rédigée sur *une copie séparée*. L'énoncé est composé de 4 pages. L'examen dure 3 heures. Les notes de cours manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.

1 Exercices

($\bullet$)Peut-on fabriquer des termes clos avec les types suivants ? Pour ceux ou c'est le cas, connait-on leur forme normale ?

$t_1 : \pi_1(\Sigma x : nat.x = 3)$
$t_2 : \pi_1(\pi_2(\Sigma x : nat.\Sigma y : nat.x.y = 5 \wedge x > y))$
$F : \Pi f : nat \to nat.(\Sigma x : nat.(f x) = 0) \vee (\Pi x : nat.(f x) = 0 \to \bot)$

($\bullet$)On est en logique du premier ordre avec les symboles de l'arithmétique $(0, S, +, \times, =)$ et les axiomes correspondants.

On se donne en plus un prédicat ternaire sur les entiers naturels, ainsi que les règles de réécriture suivante sur les propositions :

$$Q(n,q,0) \;\to\; 0 = 0 \quad (1)$$
$$Q(n,q,S(p)) \;\to\; n \neq q \times S(p) \wedge Q(n,q,p) \quad (2)$$

On rappelle qu'on identifie les propositions modulo réécriture.

En langage mathématique usuel, à quoi est équivalent $Q(n,p,n)$ lorsque $1 < p < n$?

($\bullet$)On se donne également une relation binaire R et les règles suivantes :

$$R(n,S(p)) \;\to\; R(n,p) \wedge Q(n,S(p),n) \wedge p \neq S(0) \quad (3)$$
$$R(n,0) \;\to\; 0 = 0 \quad (4)$$

Quelle est la propriété mathématique qui correspond à $R(n, n-1)$?

($\bullet$)Transposez ces définitions en Coq en définissant des prédicats vérifiant ces réécriture.

($\bullet$)Décrivez le terme preuve de $R(3,2)$. Comment croit la taille des preuves de $R(n, n-1)$ avec n ?

2 Problème 2 : Expressions Régulières et Automates

2.1 Expressions Régulières

Soit un alphabet $\mathcal{A}$ comportant au moins deux lettres a et b :

```
Parameter (A:Set) (a b:A).
```

(*) Donner un type inductif permettant de représenter les expressions régulières (*regexp*) suivantes :

– le mot vide ϵ,
– une lettre (a, b, etc.),
– la concaténation de 2 regexps ($r_1.r_2$),
– l'alternative ($r_1|r_2$),
– la répétition (r^*).

Les conventions syntaxiques dans l'énoncé sont les suivantes : la répétition a une priorité plus forte que la concaténation, qui elle-même a une priorité plus forte que l'alternative. Par exemple, $ab|b^*$ se comprend comme $(ab)|(b^*)$

(*) Définir en Coq une regexp représentant $b(ba)^*a$.

(*) Y a-t-il une façon unique de représenter une regexp ? Si oui, expliquer pourquoi, sinon donner un contre-exemple.

(*) Définir r^+, la répétition non vide (1 fois ou plus) de la regexp r, et $r?$ la regexp optionnelle (0 ou 1 fois).

(*) Écrire une fonction de type `regexp->bool` indiquant si une regexp reconnaît la chaîne vide.

(*) Définir une relation inductive caractérisant les mots (codés comme des listes de lettres) reconnus par une regexp.

```
Inductive regexp_sem : regexp -> list A -> Prop := ...
```

Rappel : le type des listes est défini ainsi :

```
Inductive list (A:Set) : Set := nil | cons (x:A) (l:list A).
```

Étant données deux listes `l1` et `l2`, la concaténation de `l1` et `l2` est notée `l1++l2`.

(*) Écrire une fonction qui construit un mot reconnu par une regexp donnée en argument.

(**) On souhaite caractériser le langage (un ensemble de mots) reconnu par une regexp à l'aide de la structure d'arbre (à branchement potentiellement infini) suivante :

```
Inductive dict : Set :=
    Node : bool -> (A->dict) -> dict
  | Fail : dict.
```

Chaque nœud de l'arbre porte un booléen indiquant si le mot formé par les lettres rencontrées en allant de la racine jusqu'au nœud appartient au langage. `Fail` indique l'absence de mot du langage ayant comme préfixe le mot formé des lettres depuis la racine.

(**) Existe-t-il, pour chaque regexp, un tel arbre représentant son langage ? Justifier la réponse.

(***) Écrire une fonction construisant l'arbre correspondant au sous-langage des mots de longueur au plus `n`, reconnus par la regexp `r`. Pour cela, on supposera qu'il existe une fonction `is_word` décidant si un mot appartient au langage d'une regexp. Puis on écrira une fonction `regexp_dict_rec` ayant un argument supplémentaire `w` indiquant le mot préfixe du langage à construire.

```
Parameter is_word : regexp -> list A -> bool.
Fixpoint regexp_dict_rec (r:regexp) (w:list A) (n:nat) {struct ...} := ...
Definition regexp_dict n r := regexp_dict_rec r nil n.
```

(***) Modifier *légèrement* la définition de `dict` permettant de représenter le langage d'une regexp. Le constructeur `Fail` est-il toujours nécessaire ? Écrire la fonction de construction du langage d'une regexp. Ici encore il faudra un paramètre supplémentaire `w` correspondant au mot préfixe du langage à construire.

2.2 Automates

Le sujet de cette section est de formaliser des automates non-déterministes à l'aide du type co-inductif suivant :

```
CoInductive automate : Set :=
  Step (fin:bool) (trans:list(option A * automate)).
```

Rappel : le type `option` est défini ainsi :

```
Inductive option (A:Set) : Set :=
  Some : A -> option A
| None : option A.
```

L'idée est qu'un objet de type `automate` représente l'automate dans un certain état. Les transitions possibles sont définies par l'argument `trans` : si cette liste contient un élément `(Some a, aut)`, alors il y a une transition vers l'automate dans l'état défini par `aut`, et la lettre `a` est consommée. Par commodité, il existe aussi des ϵ-transitions : si `trans` contient `(None, aut)` alors la transition vers `aut` se fait sans consommer de lettre. L'argument `fin` indique si l'automate est dans un état acceptant. On dira qu'un automate reconnaît un mot s'il existe un chemin de transitions vers un état acceptant consommant toutes les lettres du mot.

(*) Écrire un prédicat inductif caractérisant le langage reconnu par un automate.

```
Inductive auto_sem : automate -> list A -> Prop := ...
```

(**) Définir un automate reconnaissant le même langage que a^*b^*. Pour cela, on se choisira un ensemble S d'états (par exemple les entiers naturels `nat`). Ensuite, on écrira une fonction de type $S \rightarrow$ `automate` qui construit l'automate dans chacun de ses états. Enfin, il suffira d'appliquer cette fonction à l'état choisi comme étant l'état initial.

(***) Quelle différence cela ferait-il si le prédicat `auto_sem` avait été défini avec les mêmes constructeurs, mais de manière co-inductive ? En particulier, est-ce que cela aurait pu changer l'ensemble de mots reconnus ? Justifier ou donner un contre-exemple.

3 Problème 3 : Diamètre d'un arbre binaire

Soit G un graphe d'ensemble de sommets V. Pour deux sommets $a, b \in V$ on définit la *distance* $d(a, b)$ comme étant la longueur du plus court chemin de a à b. Le *diamètre* de G est défini comme étant la distance maximale dans le graphe, i.e.

$$\mathrm{diam}(G) := \max_{a,b \in V} d(a, b)$$

Soit maintenant T un arbre binaire. On peut le voir en particulier comme un graphe (non orienté) et donc parler de son diamètre. Dans ce problème, on se propose de spécifier et de définir une fonction calculant le diamètre d'un arbre binaire.

3.1 Graphes et chemins

Soit `A:Set` un type. On définit un graphe sur des sommets de type `A` comme un prédicat `g:A->A->Prop` représentant les arêtes de ce graphe. On suppose `A` et `g` fixés pour le reste de cette section.

☐ Définir un prédicat inductif `path:A->A->nat->Prop` tel que `path x y n` signifie « il existe un chemin de `x` à `y` de longueur `n` ». On ne cherchera pas à définir de chemin sans cycle.

☐ Énoncer un lemme indiquant que si `g` est symétrique (i.e. le graphe est non orienté) alors `path` l'est également.

☐ Définir un prédicat `dist:A->A->nat->Prop` tel que `dist x y n` signifie « la distance de `x` à `y` est `n` ».

☐ Définir un prédicat `diameter:nat->Prop` tel que `diameter n` signifie « le graphe `g` a pour diamètre `n` ».

3.2 Positions dans un arbre binaire

On se donne maintenant un type `tree:Set` des arbres binaires :

```
Inductive tree : Set := Empty : tree | Node : tree -> tree -> tree.
```

Étant donné un arbre t, tout sous-arbre de t peut être dénoté par un chemin depuis la racine de t. Un tel chemin peut être représenté par une liste de directions valant `Left` (aller à gauche) ou `Right` (aller à droite). On introduit donc les deux types Coq

```
Inductive dir : Set := Left : dir | Right : dir.
Definition pos := list dir.
```

pour représenter les positions dans un arbre.

☐ Définir un prédicat `valid:tree->pos->Prop` signifiant qu'une position est valide pour un arbre i.e. désigne effectivement l'un de ses sous-arbres.

☐ Définir un prédicat `adj:pos->pos->Prop` tel que `adj p1 p2` signifie « le sous-arbre désigné par `p1` est un sous-arbre *immédiat* du sous-arbre désigné par `p2`, ou inversement ». (On ne se préoccupe pas ici de savoir si ces deux positions sont valides pour un arbre.)

☐ Définir enfin un prédicat `gt:tree->pos->pos->Prop` tel que `gt t p1 p2` signifie « les deux positions `p1` et `p2` sont valides pour `t` et l'une représente un sous-arbre immédiat de l'autre ». En déduire la notion de diamètre d'un arbre binaire.

3.3 Calcul du diamètre d'un arbre binaire

☐ Soient t_1 et t_2 deux arbres. Montrer que

$$\mathrm{diam}(\texttt{Node}\ t_1\ t_2) = \max(\mathrm{diam}(t_1),\ \mathrm{diam}(t_2),\ 2 + h(t_1) + h(t_2))$$

où $h(t)$ désigne la hauteur de l'arbre t (la feuille `Empty` est considérée de hauteur 0). On ne demande pas une preuve Coq.

☐ En déduire un algorithme pour calculer le diamètre d'un arbre binaire en temps linéaire par rapport à son nombre de nœuds. Indication : calculer la hauteur en même temps que le diamètre.

☐ Écrire et spécifier une fonction Coq calculant le diamètre d'un arbre binaire avec cet algorithme. (On pourra au choix définir une fonction purement informative et énoncer sa correction par un lemme, ou définir une fonction fortement spécifiée.) On ne demande pas de faire la preuve de correction.