Examen du cours IN303

Correction.

On définit un *pré-tas* comme étant un arbre dont les nœuds sont étiquetés par des entiers et ont deux fils, un fils gauche et un fils droit :

```
Inductive pretas : Set := V : pretas | B : pretas -> nat -> pretas -> pretas.
```

`V` est le pré-tas vide, et `(B g n d)` est le pré-tas de fils gauche `g`, de fils droit `d` et de valeur `n`. Les pré-tas servent de structure de stockage des valeurs entières à chaque nœud.

Un *tas* est un pré-tas vide ou bien un pré-tas `(B g n d)` de valeur `n` inférieure ou égale à toutes les valeurs stockées dans `g` ou `d`, et tel que `g` et `d` sont des tas.

Le but de l'exercice est de formaliser certaines propriétés des tas; une des applications en est le *tri par tas* (heapsort).

1. Le type `pretas` est inductif, ce qui signifie que l'ensemble des pré-tas est le plus petit tel que `V` soit un pré-tas et `(B g n d)` est un pré-tas pour tout entier `n` dès que `g` et `d` sont des pré-tas. Ceci est représenté en Coq par un principe de récurrence. Dites lequel :

   (A)
   ```
   (P:liste->Prop) (P nil) -> ((l:liste) (n:nat) (P l) -> (P (cons n l)))
                             -> (l:liste) (P l)
   ```
   (B)
   ```
   (P:pretas->Prop) (P V) ->
                    ((g:pretas)  (P g) -> (n:nat) (d:pretas) (P d)
                          -> (P (B g n d)))
                      -> (t:pretas) (P t)
   ```
   (C)
   ```
   (P:pretas->Prop) (P V) ->
                    ((d:pretas) (n:nat) (P d) -> (g:pretas) (P (B g n d)))
                      -> (t:pretas) (P t)
   ```

   (B). La formule (A) est le principe de récurrence des listes d'entiers, et (C) est n'importe quoi.

2. On veut écrire un prédicat `minore` tel que `(minore n t)` soit vrai si et seulement si l'entier `n` est inférieur ou égal à tous les entiers stockés dans le pré-tas `t`. Parmi les possibilités suivantes, quelle est la seule bonne ? (On suppose comme dans l'exemple du tri par insertion qu'on dispose d'une fonction `au_plus : nat -> nat -> bool` telle que `(au_plus n m)` est `true` ssi `n`$\leq$`m`.)

   (A)
   ```
   Fixpoint minore [n:nat; t:pretas] : Prop :=
     Cases t of
         V => True
       | (B g m d) => (au_plus n m)=true /\ (minore n g) /\ (minore n d)
     end.
   ```

(B)
```
Fixpoint minore [n:nat; t:pretas] : Prop :=
   Cases t of
       V => False
     | (B g m d) => (au_plus n m)=true /\ (minore n g) /\ (minore n d)
   end.
```

(C)
```
Fixpoint minore [n:nat; t:pretas] : Prop :=
   Cases t of
       V => False
     | (B g m d) => (au_plus n m)=true /\ (minore n t)
   end.
```

(A). (B) et (C) sont erronés, ne serait-ce que parce qu'ils précisent que `(minore n V)` *doit être faux, alors que tout entier minore l'arbre vide : en effet, minorer signifie être inférieur ou égal à tout élément de l'arbre, et il n'y a pas d'élément dans* `V`*.*

3. On veut maintenant définir un prédicat `tas` tel que `(tas t)` soit vrai ssi le pré-tas `t` est un tas. Lequel proposez-vous ?

(A)
```
Fixpoint tas [t:pretas] : Prop :=
   Cases t of
       V => True
     | (B g n d) => (minore n t) /\ (tas g d)
```

(B)
```
Fixpoint tas [t:pretas] : Prop :=
   Cases t of
       V => True
     | (B g n d) => (minore n g) /\ (tas g) /\ (minore n d) /\ (tas d)
   end.
```

(C)
```
Fixpoint tas [t:pretas] : bool :=
   Cases t of
       V => True
     | (B g n d) => (minore n t) /\ (tas g) /\ (tas t)
   end.
```

(B). (A) est syntaxiquement incorrect à cause de `(tas g d)`*, et (C) est incorrect car la récurrence sur* `t` *n'est pas bien fondée.*

4. Quelle est la bonne lecture de la ligne suivante ?

```
Lemma choix_bool : (b:bool) b=true \/ b=false.
```

(A) `b` est un booléen qui est soit vrai soit faux.

(B) Tout booléen est vrai ou alors `b` est faux.

(C) Tout booléen est soit vrai soit faux.

(C).

5. Montrer informellement le lemme :

```
Lemma ajoute_minore : (t:pretas) (m,n:nat) (au_plus m n)=true -> (minore m t)
                          -> (minore m (ajoute n t)).
```

Vous pourrez vous aider des lemmes auxiliaires suivants, sans avoir à les démontrer :

```
Lemma choix_bool : (b:bool) b=true \/ b=false.
Lemma au_plus_trans : (m,n,k:nat) (au_plus m n)=true -> (au_plus n k)=true ->
                            (au_plus m k)=true.
```

Vous préciserez soigneusement quels dispositifs logiques vous utilisez : récurrences [sur quoi], analyses de cas, lemmes utilisés, etc.

(La définition de `ajoute` *est en question 6, désolé pour l'erreur.)*

Par récurrence sur t :

(a) *Si* `t` *est le pré-tas vide* `V`*, alors* `(minore m (ajoute n t))`=`(minore m (B V n V))` =`(au_plus m n)=true /\ True /\ True`*.* `(au_plus m n)=true` *est vrai par hypothèse, et le reste est trivial.*

(b) *Sinon,* `t` *est de la forme* `(B g k d)`*, et par récurrence sur* `g` *et* `d` *respectivement :*

```
Hg : (m,n:nat) (au_plus m n)=true->(minore m g)->(minore m (ajoute n g))
Hd : (m,n:nat) (au_plus m n)=true->(minore m d)->(minore m (ajoute n d))
```

On souhaite prouver que, sous les hypothèses `(au_plus m n)=true` *et* `(minore m t)`*, la conclusion* `(minore m (ajoute n t))` *tient. Supposons donc vraies ces deux hypothèses. En particulier,* `(minore m t)` *implique* `(au_plus m k)=true`*,* `(minore m g)` *et* `(minore m d)`*.*

Comme `t`=`(B g k d)`*, il nous reste à prouver :*

```
(minore m
   if (au_plus n k)
   then (B (ajoute k d) n g)
   else (B (ajoute n d) k g))
```

Or `(au_plus n k)` *est soit vrai soit faux par* `choix_bool`*, donc on a deux cas :*

i. *Si* `(au_plus n k)` *est vrai, alors par transitivité de* `au_plus` *(autrement dit par le lemme* `au_plus_trans`*) et en utilisant le fait que* `(au_plus m n)=true`*, on déduit que* `(au_plus m k)` *est vrai. Comme* `(minore m d)`*, on en déduit par récurrence sur* `d`*, c'est-à-dire en appliquant* `Hd` *ci-dessus, que* `(minore m (ajoute k d))`*. D'autre part,* `(au_plus m n)=true` *et* `(minore m d)` *sont vraies, donc* `(minore m (B (ajoute k d) n g))`*.*

ii. *Sinon, par récurrence sur* `d` *encore (en utilisant* `Hd` *encore), et parce que* `(au_plus m n)=true` *et que* `(minore m d)`*, on déduit* `(minore m (ajoute n d))`*. Comme* `(au_plus m k)=true` *et* `(minore m g)`*, on en déduit* `(minore m (B (ajoute n d) k g))`*.*

Dans les deux cas, on a le résultat souhaité.

6. Montrer, de la même façon, le lemme :

```
Lemma au_plus_minore : (t:pretas) (m,n:nat) (au_plus m n)=true -> (minore n t)
                          -> (minore m t).
```

Par récurrence sur `t` *encore :*

(a) *Si* `t=V`, *alors* `(minore m t)` *est trivial.*

(b) *Sinon,* `t` *est de la forme* `(B g k d)`, *et les hypothèses de récurrence sur* `g` *et* `d` *sont :*

```
Hg : (m,n:nat)(au_plus m n)=true->(minore n g)->(minore m g)
Hd : (m,n:nat)(au_plus m n)=true->(minore n d)->(minore m d)
```

On souhaite prouver que, sous les hypothèses `(au_plus m n)=true` *et* `(minore n t)` *(cette dernière impliquant les propriétés* `(au_plus n k)=true`, `(minore n g)` *et* `(minore n d)`*), le résultat* `(minore m t)` *tient. On a donc trois sous-buts à démontrer :*

i. `(au_plus m k)=true` *: par* `au_plus_trans`, *en utilisant le fait que* `(au_plus m n)=true` *et* `(au_plus n k)=true` *par hypothèse.*

ii. `(minore m g)` *: par hypothèse de récurrence* `Hg`.

iii. `(minore m d)` *: par hypothèse de récurrence* `Hd`.

7. On définit une fonction `ajoute` telle que `(ajoute n t)` ajoute un entier `n` au tas `t` comme suit :

```
Fixpoint ajoute [n:nat; t:pretas] : pretas :=
  Cases t of
      V => (B V n V)
    | (B g m d) => if (au_plus n m)
                   then (B (ajoute m d) n g)
                   else (B (ajoute n d) m g)
  end.
```

Montrer informellement que :

```
Lemma ajoute_preserve_tas : (t:pretas) (tas t) -> (n:nat) (tas (ajoute n t)).
```

Vous pourrez vous aider des lemmes démontrés précédemment, ainsi que du lemme auxiliaire suivant que vous n'aurez pas besoin de démontrer :

```
Lemma au_plus_total : (m,n:nat) (au_plus m n)=false -> (au_plus n m)=true.
```

Encore une fois par récurrence sur `t` *:*

(a) *Si* `t=V`, *alors* `(tas (ajoute n t))` *est égal à* `(tas (B V n V))`, *c'est-à-dire à* `(minore n V) /\ (tas V) /\ (minore n V) /\ (tas V)`, *autrement dit à* `True/\True/\True/\True`, *qui est trivial.*

(b) *Sinon,* `t` *est de la forme* `(B g k d)`, *avec les hypothèses de récurrence :*

```
Hg : (tas g)->(n:nat)(tas (ajoute n g))
Hd : (tas d)->(n:nat)(tas (ajoute n d))
```

On souhaite maintenant montrer que, sous l'hypothèse `(tas t)`*, le résultat* `(tas (ajoute n t))` *tient. L'hypothèse implique les faits* `(minore k g)`*,* `(tas g)`*,* `(minore k d)` *et* `(tas d)`*. Il reste alors à montrer :*

```
(tas
   if (au_plus n k)
   then (B (ajoute k d) n g)
   else (B (ajoute n d) k g))
```

Par `choix_bool`*, on peut considérer deux cas :*

i. *Si* `(au_plus n k)` *est vrai, il reste à montrer* `(tas (B (ajoute k d) n g))`*, c'est-à-dire les quatre conclusions suivantes :*
   - *A.* `(minore n (ajoute k d))` *: comme* `(au_plus n k)` *est vrai et* `(minore k d)` *par hypothèse, par le lemme* `au_plus_minore` *on infère* `(minore n d)`*; comme en plus* `(au_plus n k)` *est vrai, on en conclut* `(minore n (ajoute k d))` *par le lemme* `ajoute_minore`*.*
   - *B.* `(tas (ajoute k d))` *: par hypothèse de récurrence* `Hd`*, en utilisant le fait que* `(tas d)`*.*
   - *C.* `(minore n g)` *: par* `au_plus_minore`*, en utilisant* `(au_plus n k)=true` *et* `(minore k g)`*.*
   - *D.* `(tas g)` *: par hypothèse.*

ii. *Sinon,* `(au_plus n k)` *est faux, donc par* `au_plus_total` *le booléen* `(au_plus k n)` *est vrai. Il reste à montrer* `(tas (B (ajoute n d) k g))`*, c'est-à-dire les quatre propriétés suivantes :*
   - *A.* `(minore k (ajoute n d))` *: par le lemme* `ajoute_minore`*, puisque* `(au_plus k n)` *est vrai et* `(minore k d)` *par hypothèse.*
   - *B.* `(tas (ajoute n d))` *: par hypothèse de récurrence* `Hd`*, en utilisant le fait* `(tas d)`*.*
   - *C.* `(minore k g)` *: par hypothèse.*
   - *D.* `(tas g)` *: par hypothèse.*

8. (Question bonus.) Spécifier un prédicat `memes_elements : liste -> pretas -> Prop` disant qu'une liste d'entier et un pré-tas contiennent les mêmes éléments, où :

```
Inductive liste : Set := nil : liste | cons : nat -> liste -> liste.
```

et montrer que la fonction de conversion d'une liste en tas :

```
Fixpoint conversion [l:liste] : pretas :=
  Cases l of
      nil => V
    | (cons n l') => (ajoute n (conversion l'))
  end.
```

produit d'une part un tas :

```
Lemma conversion_produit_tas : (l:liste) (tas (conversion l)).
```

et d'autre part produit un tas ayant mêmes éléments que la liste de départ :

```
Lemma conversion_memes_elements : (l:liste) (memes_elements l (conversion l)).
```

Voir fichier Coq fourni.

L'idée du tri par tas est de procéder comme suit : soit `l` une liste d'entiers à trier. Le tas résultant `(conversion l)`, étant un tas, possède à sa racine l'élément minimum de la liste `l` (à moins que `l` ne soit vide). Il ne reste plus qu'à éliminer la racine de ce tas pour obtenir le tas des éléments restants de `l`; en itérant ce processus, on obtient tous les éléments de `l` en ordre croissant.