Arbres binaires

Arbres binaires

On travaille avec des arbres binaires munis d'étiquettes au niveau des nœuds internes. Une étiquette sera représentée par un élément de type `nat`.

1. Définir un type inductif `btree` permettant de décrire de tels arbres. Consulter le principe d'induction structurelle associé, appelé `btree_ind`.
2. Écrire une fonction `mirror` qui échange récursivement les fils gauche et fils droit de chaque noeud.
3. Proposer une démonstration du fait que la fonction `mirror` est idempotente, c'est-à-dire que `forall a: btree, mirror (mirror a) = a`. (Indication : `rewrite`.)
4. Écrire une fonction `tmap`, de type `(nat → nat) → btree → btree`, permettant d'appliquer une même opération à chaque étiquette de l'arbre.
5. Proposer une manière de démontrer que `forall (f g: nat→nat) (a: btree),`
   `tmap f (tmap g a) = tmap (fun x => f (g x)) a`
6. Écrire une fonction `btree_to_list` permettant de transformer un arbre en liste, où les éléments sont placés dans l'ordre du parcours infixe. (Indication : utiliser `::` pour `cons`, `++` pour `append`.)
7. Énoncer formellement en Coq que : étant donné une fonction `f: nat → nat`, appliquer la fonction `btree_to_list` sur un arbre, puis l'opération `map` sur la liste obtenue revient au même que d'appliquer `tmap` sur ce même arbre, puis de le transformer en liste avec `btree_to_list`. (Indication : lemme `map_app`.)
8. Définir une fonction `nb_labels` qui compte le nombre d'étiquettes d'un arbre.
9. Définir une fonction `height` qui permet de calculer la hauteur d'un arbre binaire. (Indication : utiliser la fonction `max` du module `Max`.)
10. Prouver que la hauteur est toujours plus petite que le nombre d'étiquettes. (Indication : tactique `case_eq` avec lemme `max_dec`, lemmes `plus_comm`, `le_plus_trans`.)
11. Soit la fonction `btree_in` (où `Leaf` et `Node` sont les constructeurs de `btree`) :
    ```
    Fixpoint btree_in (x: nat) (t: btree) : Prop :=
    match t with
    | Leaf => False
    | Node l e r => (btree_in x l) \/ e=x \/ (btree_in x r)
    end.
    ```
    Prouver que : `forall x t, btree_in x t -> In x (btree_to_list t).`
    (Indication : lemme `in_or_app`, tactique `destruct H as [Ha | [Hb | ...]]`.)
12. Prouver le lemme suivant :
    ```
    Lemma btree_in_mirror_1: forall t x, btree_in x t -> btree_in x (mirror t).
    ```
    Puis, *sans utiliser l'induction*, prouver la réciproque, à savoir :
    ```
    Lemma btree_in_mirror_2: forall t x, btree_in x (mirror t) -> btree_in x t.
    ```

Arbres binaires de recherche

On considère maintenant les arbres binaires définis ci-dessus comme des arbres binaires de recherche (*binary search trees*, ou *BST*), toujours étiquetés par des entiers `nat`. On rappelle qu'un arbre binaire de recherche est un arbre pour lequel l'étiquette d'un nœud interne est supérieure à toutes les étiquettes présentes dans son sous-arbre gauche et inférieure à toutes les étiquettes présentes dans son sous-arbre droit. On suppose que les arbres binaires de recherche ne contiennent pas de doublons, c'est-à-dire que « supérieur » et « inférieur » sont à prendre au sens strict.

13. Définir un prédicat inductif `btree_lt`, de type `btree` → `nat` → `Prop`, qui exprime que toutes les étiquettes d'un arbre sont (strictement) inférieures à un entier donné. Définir symétriquement un prédicat inductif `btree_gt`.
14. Prouver les lemmes :
```
Lemma btree_in_lt: forall t x n, btree_in x t -> btree_lt t n -> x < n.
Lemma btree_in_gt: forall t x n, btree_in x t -> btree_gt t n -> n < x.
```
(Indication : `inversion`, souvent suivie de `subst`.)
15. Définir un prédicat inductif `bst` exprimant qu'un arbre est un arbre binaire de recherche.
16. Définir une fonction `bst_in` qui teste si un entier est présent dans un arbre binaire de recherche, en mettant à profit l'organisation de l'arbre. (Indication : utiliser la syntaxe `if-then-else`, avec les lemmes `eq_nat_dec`, `lt_dec`.)
17. Prouver les lemmes :
```
Lemma bst_btree_in: forall x t, bst_in x t -> btree_in x t.
Lemma btree_bst_in: forall x t, bst t -> btree_in x t -> bst_in x t
```
(Indication : simplifiez les expressions avec `case_eq (...); intros`.)
18. Définir une fonction d'insertion `bst_insert` (de type `btree` → `nat` → `btree`) dans un arbre binaire de recherche.
19. Prouver qu'après insertion, un arbre binaire de recherche est toujours un arbre binaire de recherche. Nous vous proposons la démarche suivante, qui consiste à prouver deux lemmes annexes :
```
Lemma btree_lt_insert: forall t n x,
    bst t -> btree_lt t n -> x < n -> btree_lt (bst_insert t x) n.
Lemma btree_gt_insert: forall t n x,
    bst t -> btree_gt t n -> n < x -> btree_gt (bst_insert t x) n.
```
Le lemme demandé est :
```
Lemma bst_insert_bst: forall t n, bst t -> bst (bst_insert t n).
```

99. Prouver que :
```
Lemma bst_to_list_sorted: forall t, bst t -> sorted (btree_to_list t).
```
où `sorted` est une prédicat inductif (à définir) exprimant le fait que les éléments de la liste sont triés par ordre croissant.