Sujet 1 (02/10/2024)

contact : jean-baptiste.doderlein@ens-rennes.fr

Question de cours

Rappeler le principe d'induction.

Exercice 1 : Types en OCaml

Donner le type (s'il existe) de `f`, `g`, `h` et `p` :

```
let f x y l = (x+.y)::l

let g x = x x

let h x y =
    let hp x = (x+.y)>6. in
    let hpp y = x::y in
    hpp [hp 5.]

let p =
    let pp x y = x@y in
    pp [1.;2.]
```

Exercice 2 : Arbre binaire strict

On définit l'ensemble des arbres strictement binaires (B) par :

- une feuille F est un arbre strictement binaire.
- Si a et b sont deux arbres strictement binaires, alors $N(a, b)$, appelé noeud, est un arbre strictement binaire.

1. Définir par induction la fonction *taille* : $B \to \mathbb{N}$ qui donne le nombre de noeud d'un arbre strictement binaire.

2. Faire de même avec la hauteur d'un arbre strictement binaire.

3. Combien y a t'il d'arbres strictement binaires avec 2 feuilles ? et à 3 feuilles ?

4. Montrer que le nombre d'arbres strictement binaires de $n+1$ feuilles est égal à C_n définie par :

- $C_0 = 1$
- $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$ pour $n \geq 0$

5. Proposer un type OCaml `asb` pour représenter des arbres strictement binaires.

6. On appelle un arbre parfait, un arbre dont toutes les feuilles sont à la même distance de la racine. Écrire une fonction `parfait : int -> asb` qui prend un entier n et renvoie un arbre strictement binaire parfait de hauteur n.

7. Écrire une fonction `est_parfait : asb -> bool` qui renvoie vrai si un arbre strictement binaire est parfait.

Sujet 2 (02/10/2024)

contact : jean-baptiste.doderlein@ens-rennes.fr

Question de cours

Rappeler le principe de récurrence forte.

Exercice 1 : Entiers de Peano

On définit inductivement les entiers de Peano (notés P) par :

- Zéro (noté Z) est un entier de Peano.
- Si x est un entier de Peano, alors $S(x)$, appelé successeur de x, est un entier de Peano.

Par exemple, $S(S(Z))$ est un entier de Peano, et représente l'entier 2.

1. Comment représenter 4 avec les entiers de Peano ?

2. Définir par induction une fonction d'évaluation *eval* : $P \to \mathbb{N}$ tel que *eval(x)* est l'entier dans $\mathbb{N}$ représenté par x.

3. Définir inductivement sur le premier argument la fonction *plus:* $P \times P \to P$

On définit l'ensemble P' par :

- Zéro (noté Z) appartient à P'
- Si x appartient à P', alors $S(S(x))$ appartient à P'

4. Montrer que $P' \subseteq P$.

5. Que représentent les éléments de P' ?

6. Montrer que P' est stable par addition.

7. Définir le type OCaml des entiers de Peano `peano`.

8. Écrire `est_pair : peano -> bool`.

9. Écrire `convert : int -> peano`.

10. Écrire une fonction `somme : peano list -> peano` qui renvoie la somme d'une liste d'entiers de Peano.

Exercice 2 : Comptage dans une liste

1. Écrire une fonction `assoc : 'a -> ('a * 'b) list -> 'b option` qui renvoie l'élément associé à l'élément de type `'a` de la liste de couples s'il existe.

2. Écrire une fonction `set : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list` qui modifie l'élément `'b` associé à l'élément `'a` de la liste de couples. Cette fonction doit insérer le couple s'il n'existe pas.

3. Écrire une fonction `maxi : int list -> int` qui renvoie le plus grand entier de la liste passée en argument.

4. Écrire une fonction `count : int list -> int -> int` qui, pour une liste d'entier l et un entier n passés en argument, renvoie le nombre d'occurrences de n dans l.

5. Écrire une fonction de type `int list -> (int * int) list` qui pour une liste de nombres renvoie une liste de couples (entier, nombre d'occurrences).

Sujet 3 (02/10/2024)

contact : jean-baptiste.doderlein@ens-rennes.fr

Question de cours

Rappeler et démontrer le principe de récurrence simple.

Exercice 1 : Concatenation de listes

On définit inductivement les listes (L) par :

- `[]` est une liste.
- Si `h` est un élément et `t` est une liste, alors `h::t` est une liste.

1. Définir par induction la fonction *longueur* : $L \to \mathbb{N}$.

2. Définir par induction la concaténation de listes *concat* : $L \times L \to L$.

3. Montrer que la concaténation de deux listes l_1 et l_2 produit une liste dont la longueur est $l_1 + l_2$.

4. Montrer que la concaténation est associative.

5. Écrire en OCaml la fonction `append : 'a list -> 'a list -> 'a list` qui réalise la concaténation de deux listes.

6. Écrire en OCaml la fonction `flatten : 'a list list -> 'a list` qui réalise la concaténation d'une liste de listes.

Exercice 2 : Stream

On introduit un nouveau type, les stream. On les définit d'une manière similaire aux listes chainées d'OCaml avec le type :

```
type 'a stream =
| Nil
| Cons of 'a * (unit -> 'a stream)
```

1. Rappeler le type `list` de OCaml. Quelles différences avec les stream ?

2. Écrire la fonction `singleton : 'a -> 'a stream` qui renvoie un stream qui contient un unique élément.

3. Écrire la fonction `cons : 'a -> 'a stream -> 'a stream` qui ajoute un élément en tête du stream.

4. Écrire la fonction `uncons : 'a stream -> ('a * 'a stream) option` qui renvoie l'élément en tête et le reste de la liste.

5. Écrire la fonction `stream_to_list : 'a stream -> 'a list` qui convertit un stream en liste OCaml.

6. Écrire la fonction `list_to_stream : 'a list -> 'a stream` qui convertit une liste OCaml en un stream.

7. Soit la fonction `entiers` définie ainsi :

```
let rec entiers n =
    let suite () = entiers (n+1) in
    Cons(n, suite)
```

Que fait cette fonction ? Quel serait le résultat de `uncons (entiers 10)` ? et `stream_to_list (entiers 1)` ?