Examen blanc

Programmation IV

26 mai 2004

Nom : _____

Prénom : _____

Section : _____

Exercice	Points	Points obtenus
1	7	
2	9	
3	7	
4	8	
5	12	
Total	43	

Exercice 1 : Fonctions d'ordre supérieur (7 points)

On rappelle la définition de la méthode `foldRight` dans la classe `List` :

```
class List[a] {
  def foldRight[b](z: b)(f: (a, b) => b): b = match {
    case Nil => z
    case x :: xs => f(x, xs.foldRight(z)(f))
  }
  ...
}
```

1) Expliquez ce que calcule l'expression suivante définie en terme de `foldRight`.

```
  (List.range(0, 10) foldRight 0) { (i, acc) => i * i + acc }
```

2) Expliquez ce que calcule la fonction suivante :

```
  def foo[a](l1: List[a], l2: List[a]): List[a] = l1 match {
    case Nil => l2
    case x :: xs => x :: foo(xs, l2)
  }
```

3) Donnez une définition équivalente à la fonction `foo` qui ne soit pas récursive (en utilisant `foldRight` par exemple).

Exercice 2 : Entiers inductifs (9 points)

Voici une définition inductive des entiers :

```
trait Nat {
  def + (n: Nat): Nat = match {
    case Z    => n         //    Z + n = n
    case S(m) => S(m + n)  // S(m) + n = S(m + n)
  }
}
case object Z extends Nat;         // Zero
case class S(n: Nat) extends Nat;  // Successeur de n
```

Il y a une constante (`Z`) pour représenter l'entier 0, un constructeur unaire (`S`) pour construire le successeur d'un entier donné, et la définition de l'addition (`+`) entre deux entiers.

Étant données ces définitions, démontrez, en justifiant chaque étape, les trois propriétés suivantes :

$$
\begin{array}{lllcl}
1) & \forall m, n, p : Nat, & (m + n) + p & = & m + (n + p) \\
2) & \forall n : Nat, & n + Z & = & n \\
3) & \forall m, n : Nat, & m + S(n) & = & S(m + n)
\end{array}
$$

Exercice 3 : Décomposition en facteurs premiers (7 points)

Écrivez une fonction `primeFactors` qui renvoie la liste des facteurs premiers d'un entier $n \geq 2$ donné. Si un nombre premier divise plusieurs fois n, il doit aussi apparaître plusieurs fois dans la liste.

Voici quelques exemples :

$$\begin{array}{lcl} \mathit{primeFactors}(5) & = & \mathit{List}(5) \\ \mathit{primeFactors}(8) & = & \mathit{List}(2,2,2) \\ \mathit{primeFactors}(12) & = & \mathit{List}(2,2,3) \end{array}$$

Aide : vous pouvez, mais ce n'est pas absolument nécessaire, utiliser sans le redéfinir le flot des nombres premiers défini en cours.

Exercice 4 : Flots (8 points)

1) Écrivez une fonction `zip` qui prend en paramètres deux flots infinis et qui retourne un flot infini appariant les éléments des deux premiers flots.
Autrement dit, si on a les deux flots suivants :

$$\begin{aligned} s_1 &= (x_0, x_1, x_2, \ldots) \\ s_2 &= (y_0, y_1, y_2, \ldots) \end{aligned}$$

alors

$$zip(s1, s2) = s$$

avec

$$s = (Pair(x_0, y_0), Pair(x_1, y_1), \ldots)$$

2) Écrivez ensuite une fonction `unzip` qui réalise l'opération inverse.

Remarque : ces deux fonctions doivent êtres polymorphes, c'est-à-dire qu'elles doivent fonctionner quel que soit le type des éléments des flots.

Exercice 5 : Intervalles (12 points)

On décide dans cet exercice de représenter les ensembles finis d'entiers par des listes de paires d'entiers :

```
type Set = List[Pair[int, int]];
```

Chaque paire d'entiers $Pair(m, n)$ représente un intervalle fermé $[m, n]$. L'ensemble représenté par une liste de paires est l'union des intervalles représentés par ces paires.
Par exemple, voici comment sont représentés quelques ensembles d'entiers sous forme de listes d'intervalles.

$$\begin{array}{rcl} \emptyset & \rightarrow & List() \\ \{4,5,6\} & \rightarrow & List(Pair(4,6)) \\ \{9\} & \rightarrow & List(Pair(9,9)) \\ \{4,5,6,9\} & \rightarrow & List(Pair(4,6), Pair(9,9)) \end{array}$$

Étant donnée une liste de paires s représentant un ensemble

$$s = List(Pair(l_1, r_1), \ldots, Pair(l_n, r_n)) \qquad n \geq 0$$

on a donc les propriétés suivantes :

$$\begin{array}{ll} \forall\, i \in [1, n], & l_i \leq r_i \\ \forall\, i \in [1, n-1], & r_i < (l_{i+1} - 1) \end{array}$$

Écrivez les fonctions suivantes, définies sur les ensembles d'entiers représentés comme des listes de paires :

1) la fonction `contains` qui teste si un entier appartient à un ensemble,

2) la fonction `size` qui renvoie le nombre d'éléments appartenant à un ensemble,

3) la fonction `union` qui retourne l'union de deux ensembles.

```
def union(s1: Set, s2: Set): Set = Pair(s1, s2) match { ...
```