Programmation IV Examen intermédiaire - corrigé 11 mai 2005

Exercice 1 : Manipulation de listes (10 points)

Partie 1

La fonction `zipWith` s'écrit comme suit :

```
def zipWith[A,B,C](xs: List[A], ys: List[B], f: (A,B) => C): List[C] =
  if (xs.isEmpty || ys.isEmpty)
    Nil
  else
    f(xs.head, ys.head) :: zipWith(xs.tail, ys.tail, f);
```

Partie 2

La fonction `zipAll` avec filtrage de motifs s'écrit comme suit :

```
def zipAll[A,B](xs: List[A], ys: List[B], x: A, y: B): List[Pair[A,B]] = {
  def aux(xs: List[A], ys: List[B]): List[Pair[A,B]] = Pair(xs, ys) match {
    case Pair(Nil, Nil)           => Nil
    case Pair(hd1::tl1, Nil)      => Pair(hd1,   y) :: aux(tl1, Nil)
    case Pair(Nil, hd2::tl2)      => Pair(  x, hd2) :: aux(Nil, tl2)
    case Pair(hd1::tl1, hd2::tl2) => Pair(hd1, hd2) :: aux(tl1, tl2)
  }
  aux(xs, ys)
}
```

La fonction `zipAll` sans filtrage de motifs s'écrit comme suit :

```
def zipAll[A,B](xs: List[A], ys: List[B], x: A, y: B): List[Pair[A,B]] =
  if (xs.isEmpty && ys.isEmpty)
    Nil
  else if (xs.isEmpty)
    List.make(ys.length, x) zip ys
  else if (ys.isEmpty)
    xs zip List.make(xs.length, y)
  else
    Pair(xs.head, ys.head) :: zipAll(xs.tail, ys.tail, x, y);
```

Exercice 2 : Preuve par induction structurelle (15 points)

Pour démontrer l'égalité

```
mapFun(append(xs, ys), f) = append(mapFun(xs, f), mapFun(ys, f))
```

on effectue une preuve par induction structurelle sur `xs`.

Pour cela on traite tout d'abord le cas de base pour chaque côté de l'égalité avant de passer à l'étape d'induction.

1. Cas de base : `xs = Nil`

   Pour le côté gauche on a :

   ```
       mapFun(append(Nil, ys), f)
   =   (selon 1ère clause de append)
       mapFun(ys, f)
   ```

   Pour le côté droit on a :

   ```
       append(mapFun(Nil, f), mapFun(ys, f))
   =   (selon 1ère clause de mapFun)
       append(Nil, mapFun(ys, f))
   =   (selon 1ère clause de append)
       mapFun(ys, f)
   ```

2. Etape d'induction : `xs = z :: zs`

   On suppose que l'égalité est vérifiée pour `zs` et on veut montrer qu'elle l'est également pour `xs` :

   $\forall$ `ys`, $\forall$ `f`, `mapFun(append(zs,ys),f) = append(mapFun(zs,f),mapFun(ys,f))`

   Pour le côté gauche on a :

   ```
       mapFun(append(z :: zs, ys), f)
   =   (selon 2ème clause de append)
       mapFun(z :: append(zs, yp), f)
   =   (selon 2ème clause de mapFun)
       f(z) :: mapFun(append(zs, ys), f)
   ```

   Pour le côté droit on a :

   ```
       append(mapFun(z :: zs, f), mapFun(ys, f))
   =   (selon 2ème clause de mapFun)
       append(f(z) :: mapFun(zs, f), mapFun(ys, f))
   =   (selon 2ème clause de append)
       f(z) :: append(mapFun(zs, f), mapFun(ys, f))
   =   (selon l'hypothese d'induction)
       f(z) :: mapFun(append(zs, ys), f)
   ```

   CQFD

Exercice 3 : Triangle de Pascal (10 points)

Partie 1

La fonction `nextRow` s'écrit comme suit :

```
def nextRow(xs: Row): Row = {
  def leftShift(xs: Row): Row = xs ::: List(0);
  def rightShift(xs: Row): Row = 0 :: xs;
  (leftShift(xs) zip rightShift(xs)) map { case Pair(x,y) => x + y };
}
```

En utilisant la fonction `zipAll` de l'exercice 1 on peut également écrire ceci :

```
def nextRow(xs: Row): Row =
  zipAll(xs, 0 :: xs, 0, 0) map { case Pair(x,y) => x + y };
```

Partie 2

La fonction `pascal` avec récursion terminale s'écrit comme suit :

```
def pascal(n: int): Triangle = {
  def aux(n: int, t: Triangle, r: Row): Triangle = {
    if (n == 0)
      t
    else
      aux(n - 1, r :: t, nextRow(r))
  }
  aux(n, List(List(1)), List(1,1))
}
```

La fonction `pascal` exprimée de façon impérative s'écrit comme suit :

```
def pascal(n: int): Triangle = {
  var t: Triangle = List(List(1));
  for (val i <- Iterator.range(0, n)) {
    t = nextRow(t.head) :: t
  }
  t
}
```

Exercice 4 : Abstraction de données (15 points)

La classe `Polynomial` s'écrit comme suit :

```
class Polynomial(xs: List[double]) extends Function[double, double] {
  val coeffs: List[double] = xs;

  def +(that: Polynomial): Polynomial = {
    val pairs = zipAll(coeffs, that.coeffs, 0.0, 0.0);
    new Polynomial(pairs map { case Pair(x, y) => x + y })
  }

  def -(that: Polynomial): Polynomial ={
    val pairs = zipAll(coeffs, that.coeffs, 0.0, 0.0);
    new Polynomial(pairs map {case Pair(x, y) => x - y })
  }

  def *(n: double): Polynomial =
    new Polynomial(coeffs map (c => c * n));

  def apply(x: double): double =
    coeffs.foldRight(0.0)((c1: double, c2: double) => c1 + x * c2);

}
```