Programmation IV Examen - corrigé 19 juin 2003

Exercice 1 : suppression d'un élément d'une liste

Dans le cas où la liste est vide, il n'y a aucun élément à supprimer, il suffit donc de retourner la liste vide. Le premier trou se remplit donc avec un simple `Nil` (ou `List()`).

Dans le cas où la liste est non vide, il y a deux possibilités : soit l'élément en tête de la liste est égal à celui à supprimer, auquel cas il suffit de retourner la queue de la liste ; sinon, il faut retourner une nouvelle liste dont la tête est la tête de l'ancienne liste et la queue est obtenue en supprimant récursivement l'élément dans la queue de l'ancienne liste.

Au final, on obtient donc la fonction suivante :

```
def removeFirst[a](lst: List[a], elem: a): List[a] = lst match {
  case Nil =>
    Nil
  case head :: tail =>
    if (head == elem) tail else head :: removeFirst(tail, elem)
}
```

Exercice 2 : multi-ensembles

Le code partiel fourni distingue deux cas : soit l'élément `head`, en tête de la liste représentant le multi-ensemble `m1`, apparaît également dans le multi-ensemble `m2`, soit il n'y apparaît pas.

Dans le premier cas, il est clair que l'élément `head`, qui apparaît à la fois dans `m1` et `m2`, fera partie du multi-ensemble résultat. Pour calculer le reste du résultat, on utilise un appel récursif en faisant bien attention à supprimer une occurrence de `head` dans `m2` auparavant. Pour ce faire, on utilise la fonction `removeFirst` de l'exercice précédent.

Dans le second cas, l'élément `head` ne fera bien entendu pas partie du résultat. Ce dernier s'obtient donc simplement avec un appel récursif sur la queue de `m1` et la totalité de `m2`.

On obtient donc finalement la définition suivante pour la fonction `intersectM` :

```
def intersectM[a](m1: List[a], m2: List[a]): List[a] = m1 match {
  case Nil => Nil
  case head :: tail =>
    if (m2 contains head)
      head :: intersectM(tail, removeFirst(m2, head))
    else
      intersectM(tail, m2)
}
```

Exercice 3 : LISP

Question 1 La fonction `f` évalue des expressions arithmétiques composées de sommes et de constantes, et représentées par des listes. Par exemple, l'expression $(1 + (2 + 3)) + 4$ est représentée par la liste LISP suivante :

```
(plus (plus (num 1) (plus (num 2) (num 3))) (num 4))
```

Le programme complet, quant à lui, utilise la fonction `f` pour évaluer l'expression $6 + (4 + 3)$.

Question 2 Un évaluateur d'expressions composées de constantes et de sommes a été vu au cours lors de la semaine 4, pp. 7–8. Il suffisait donc de reproduire ce code, éventuellement en renommant les classes cas pour qu'elles correspondent mieux à la version LISP. Le résultat est donc donné ici sans plus d'explications.

```
object Main {
  abstract class Expr;
  case class Plus(l: Expr, r: Expr) extends Expr;
  case class Num(v: int) extends Expr;

  def f(e: Expr): int = e match {
    case Plus(l, r) => f(l) + f(r)
    case Num(v) => v
  }

  def main(args: Array[String]): unit = {
    System.out.println(f(Plus(Num(6), Plus(Num(4), Num(3)))));
  }
}
```

Exercice 4 : graphes acycliques dirigés

Question 1 Comme précisé dans l'énoncé, il existe un chemin menant d'un sommet à un autre s'il existe une suite non vide d'arêtes menant du premier sommet au second.

Il existe donc d'une part un chemin d'un sommet à un autre s'il existe une arête allant du premier au second.

D'autre part, il existe un chemin allant d'un sommet à un autre s'il existe une arête menant du premier sommet à un sommet intermédiaire, puis un chemin allant de ce sommet intermédiaire au second.

En traduisant ces deux constatations en PROLOG, on obtient les deux règles suivantes pour le prédicat `pathExists` :

```
pathExists(X,Y) :- edge(X,Y).
pathExists(X,Y) :- edge(X,Z), pathExists(Z,Y).
```

Question 2 Le prédicat `path` est très similaire au prédicat `pathExists` ci-dessus, si ce n'est que le chemin doit faire partie des arguments. On peut donc reprendre les deux règles définissant `pathExists` en ajoutant le chemin comme troisième argument.

Dans le premier cas, il existe une arête menant du sommet `X` au sommet `Y`, le chemin est donc composé uniquement de ces deux sommets :

```
path(X,Y,cons(X,cons(Y,nil))) :- edge(X,Y).
```

Dans le second cas, le chemin menant du sommet `X` au sommet `Y` passe par un sommet intermédiaire `Z`, lié au sommet `X` par une arête. Le chemin final s'obtient donc en ajoutant le sommet `X` au chemin `P` menant de `Z` à `Y` :

```
path(X,Y,cons(X,P)) :- edge(X,Z), path(Z,Y,P).
```

Exercice 5 : cavalier d'échecs

Le problème du cavalier admet beaucoup de solutions, mais nous en présenterons uniquement deux ici. Toutes les deux sont récursives mais diffèrent dans leur usage de la récursion.

Toute solution récursive a besoin d'un cas de base. Pour le cavalier, ce cas de base est trivial : en au plus zéro coups, la seule position atteignable par un cavalier est celle qu'il occupe au départ. Les solutions qui suivent retournent donc toutes une liste composée de la position initiale si le nombre de coups donné est nul.

L'idée de la première solution est de procéder ainsi : pour calculer la liste des positions atteignables en au plus n coups, on commence par calculer les positions atteignables en un coup, puis récursivement toutes les positions atteignables en $n-1$ coups depuis là. Il ne faut bien entendu pas oublier d'inclure la positions originale dans le résultat, cette dernière étant toujours accessible. Le calcul des positions atteignables en un coup se fait simplement en filtrant les positions retournées par `nextPos` au moyen du prédicat `legalPos` qui s'assure que les positions sont valides, c'est-à-dire sur l'échiquier. La notation `for` de SCALA nous permet d'exprimer cela de manière concise :

```
def reachable(boardSize: int)(initial: Position, moves: int)
  : List[Position] =
  initial :: (if (moves == 0)
                Nil
              else
                for (val p <- nextPos(initial); legalPos(boardSize)(p);
                     val succ <- reachable(boardSize)(p, moves - 1))
                yield succ);
```

L'autre solution consiste à calculer récursivement les positions atteignables en $n-1$ coups depuis la position initiale, puis à calculer les positions atteignables en un coup depuis chacune d'elles. Encore une fois, il faut prendre garde à filtrer les positions retournées par `nextPos` au moyen du prédicat `legalPos` pour éviter les solutions invalides. On obtient alors le code ci-dessous.

```
def reachable(boardSize: int)(initial: Position, moves: int)
  : List[Position] =
  if (moves == 0)
    List(initial)
  else {
    val oneLess = reachable(boardSize)(initial, moves - 1);
    oneLess ::: (oneLess flatMap nextPos filter legalPos(boardSize))
  }
```

Exercice 6 : preuve par induction

Cas 1 : $l = \text{IntNil}()$

La partie de gauche se transforme ainsi :

(l map g) map f = (IntNil() map g) map f	*Valeur de l*
= IntNil() map f	*Fonction map dans IntNil*
= IntNil()	*Fonction map dans IntNil*

et la partie de droite ainsi :

l map { x ⇒ f(g(x)) } = IntNil() map { x ⇒ f(g(x)) }	*Valeur de l*
= IntNil()	*Fonction map dans IntNil*

ce qui prouve le cas de base.

Cas 2 : $l = \text{IntCons}(\text{h}, \text{t})$

La partie de gauche se transforme ainsi :

(l map g) map f = (IntCons(h,t) map g) map f	*Valeur de l*
= (IntCons(g(h), t map g)) map f	*Fonction map dans IntCons*
= IntCons(f(g(h)), (t map g) map f)	*Fonction map dans IntCons*
= IntCons(f(g(h)), t map { x ⇒ f(g(x)) }	*Hypothèse d'induction*

et la partie de droite ainsi :

l map { x⇒f(g(x)) } = IntCons(h,t) map {x⇒f(g(x))}	*Valeur de l*
= IntCons({x⇒f(g(x))}(h),t map {x⇒f(g(x))})	*map dans IntCons*
= IntCons(f(g(h)), t map {x⇒f(g(x))}	*Application de fct.*

ce qui prouve le cas général.

Exercice 7 : XML

Pour aplatir les objets XML, une simple fonction récursive suffit. Il y a deux cas à considérer :

1. les nœuds `Tagged`, dont il faut aplatir tous les fils en ignorant l'étiquette `tag`, et
2. les nœuds `Text`, qu'on aplatit en extrayant leur contenu.

La seule petite difficulté se situe au niveau des nœuds `Tagged`, puisqu'il est nécessaire de concaténer toutes les chaînes obtenues par aplatissement récursif des fils. Cela se fait aisément au moyen de la méthode `foldLeft` des listes.

```
def flatten(xml: XML): String = xml match {
  case Tagged(_, contents) =>
    contents.foldLeft("")((str: String, elem: XML) =>
                          str + flatten(elem))
  case Text(contents) => contents
}
```

Exercice 8 : flot infini

Il existe une multitude de solutions pour cet exercice également, c'est pourquoi nous n'en présenterons qu'un sous-ensemble ici. Toutes ces solutions partagent toutefois certaines idées.

La constatation principale qu'il importe de faire est que, sur une diagonale, la somme de la composante x et de la composante y d'un point est constante. Ainsi, la « diagonale » d'indice 0, qui se réduit à un point, est celle où la somme des composantes est 0. La diagonale d'indice 1 est celle où la somme des composantes est 1. De manière générale, la diagonale d'indice n est celle où la somme des composantes est n.

La première solution, peut-être la plus élégante, utilise ces constatations et la boucle `for` de SCALA pour construire le flot en deux temps : tout d'abord, le flot des indices des diagonales est construit, ce qui se fait simplement au moyen de la fonction `from` vue au cours, ce flot n'étant rien d'autre que le flot des entiers positifs ; ensuite, pour chaque diagonale, le flot des composantes y des points qui la composent est construit. La composante x s'obtient par simple calcul, puisque comme on l'a vu, la diagonale d'indice d est composée des points tels que $x + y = d$ avec $x, y \in \mathbb{N}$. On obtient donc :

```
def from(x: int): Stream[int] = Stream.cons(x, from(x + 1));

val intPairsStream =
  for (val d <- from(0); val y <- Stream.range(0, d + 1))
    yield Pair(d - y, y);
```

La seconde solution construit le flot de la manière suivante : on commence avec le point $(0, 0)$ puis, pour calculer le point suivant, on regarde si le point courant est sur l'axe des y. Si c'est le cas, il faut passer au premier point de la diagonale suivante, celui dont la première composante est nulle. Si ce n'est pas le cas, on continue avec le point suivant de la diagonale courante, situé une unité au desssus et une unité à gauche du point courant.

```
def intPairs(x: int, y: int): Stream[Pair[int,int]] =
  Stream.cons(Pair(x, y), if (x == 0) intPairs(y + 1, 0)
                          else intPairs(x - 1, y + 1));
val intPairsStream = intPairs(0,0);
```