Programmation avancée

Examen intermédiaire

vendredi 26 novembre 2010

Nom : ______________________________

Prénom : ______________________________

Vos points sont *précieux*, ne les gaspillez pas !

Votre nom Le travail qui ne peut pas vous être attribué est perdu: écrivez votre nom sur chaque feuille que vous rendez.

Votre temps Tous les points ne sont pas égaux. En effet, nous ne pensons pas que tous les exercices ont la même difficulté, même s'ils ont le même nombre de points.

Votre attention La donnée de chaque exercice est précisément formulée, et parfois subtile. Si vous ne la comprenez pas, vous ne pourrez pas en tirer tous les points.

Exercice	Points	Points obtenus
1	10	
2	10	
3	10	
Total	30	

Exercice 1 : Preuve par induction (10 points)

La méthode `zip` combine deux listes en une liste de paires. Elle est définie de la manière suivante :

```
def zip[A, B](l1: List[A], l2: List[B]): List[(A, B)] =
  (l1, l2) match {
    case (x :: xs, y :: ys) => (x, y) :: zip(xs, ys)  // (z0)
    case (Nil, xs)          => Nil                    // (z1)
    case (xs, Nil)          => Nil                    // (z2)
  }
```

La méthode `drop` ampute une liste d'un certain nombre de ses premiers éléments. Elle est définie ainsi [1] :

```
def drop[A](l: List[A], n: Int): List[A] =
  l match {
    case x :: xs if (n > 0)  => drop(xs, n - 1)  // (d0)
    case xs      if (n == 0) => xs               // (d1)
    case Nil                 => Nil              // (d2)
  }
```

Partie 1 En utilisant les définitions de `zip` et de `drop`, prouvez l'égalité suivante par induction sur `n`, en supposant que `n` ≥ 0 :

```
zip(drop(xs, n), drop(ys, n)) == drop(zip(xs, ys), n)
```

Ecrivez les hypothèses d'induction au début de tous les cas inductifs. Donnez une brève justification de chaque pas de votre preuve. Vous pouvez utiliser les noms `z0` à `z2` et `d0` à `d2` pour référencer les différents cas des fonctions `zip` et `drop`.

Suggestion : dans le cas inductif, distinguez plusieurs sous-cas en fonction de la valeur des listes `xs` et `ys`.

Partie 2 La fonction `reverse` renverse l'ordre des éléments d'une liste. Etant donné cette fonction, l'égalité suivante n'est pas vraie dans le cas général :

```
zip(reverse(xs), reverse(ys)) == reverse(zip(xs, ys))
```

Donnez un contre-exemple pour cette égalité, sous la forme de valeurs pour `xs` et `ys` qui ne la satisfont pas.

---

1. Les expressions **`if`** `(n > 0)` et **`if`** `(n == 0)` qui suivent les motifs dans les cas `d0` et `d1` sont des *gardes*, c-à-d des contraintes supplémentaires qui doivent être vraies pour que le cas auquel elles sont attacheés soit évalué. Par exemple, dans le cas `d0`, il faut non seulement que la liste `l` soit non-vide mais en plus que le paramètre `n` soit strictement positif.

Montrez, en les récrivant, que les deux côtés de l'égalité produisent un résultat différent. Pour ce faire, vous avez le droit de récrire en une seule étape les utilisations de `reverse`. Par exemple, si vous obtenez un terme de la forme

```
reverse(1 :: 2 :: 3 :: Nil)
```

vous pouvez le récrire directement en

```
3 :: 2 :: 1 :: Nil
```

en donnant comme justification « selon la définition de `reverse` ».

Exercice 2 : Ensembles d'entiers (10 points)

Un ensemble d'entiers peut être représenté par une liste triée d'intervalles fermés. Par exemple, l'ensemble $\{1, 2, 3, 5, 6, 7, 8, 9, 11\}$ peut être représenté par la liste d'intervalles $[1; 3], [5; 9], [11; 11]$.

Il existe généralement plusieurs représentations d'un ensemble donné sous forme de liste d'intervalles. Ainsi, l'ensemble ci-dessus pourrait aussi être représenté par la liste de cinq intervalles $[1; 2]$, $[2; 3]$, $[5; 7]$, $[8; 9]$ et $[11; 11]$. On définit donc une notion de *représentation canonique* d'un ensemble S sous forme de liste d'intervalles, qui est l'unique liste d'intervalles fermés $[b_1, e_1], [b_2; e_2], \ldots, [b_n; e_n]$ ayant les propriétés suivantes :

1. aucun intervalle n'est vide :
   $\forall i \in \{1, \ldots, n\}, b_i \leq e_i$,
2. les intervalles sont triés par ordre croissant et ne sont pas contigus :
   $\forall i \in \{1, \ldots, n-1\}, e_i < b_{i+1}$,
3. l'union des éléments des intervalles est l'ensemble représenté :
   $\bigcup_{i=1}^{n} \{b_i, b_i + 1, \ldots, e_i - 1, e_i\} = S$.

Par exemple, la représentation canonique de l'ensemble $\{1, 2, 3, 5, 6, 7, 8, 9, 11\}$ est la liste d'intervalles $[1; 3], [5; 9], [11; 11]$.

Dans le reste de cet exercice, vous pouvez admettre que tous les ensembles reçus par vos fonctions sont canoniques, et vous devez maintenir cette propriété pour les ensembles que vous construisez.

Partie 1 Le type `Interval` est défini comme un simple alias du type des paires d'entiers :

```
type Interval = (Int, Int)
```

Selon le même principe, définissez un type pour les ensembles d'entiers représentés par des listes de tels intervalles :

```
type IntSet = ...
```

Définissez ensuite la fonction d'appartenance pour ces ensembles, qui retourne vrai si et seulement si l'entier qu'on lui donne appartient à l'ensemble :

```
def contains(set: IntSet, item: Int): Boolean = ...
```

Votre solution n'a pas besoin d'être efficace. En particulier, il n'est pas nécessaire qu'elle tire parti du fait que les intervalles sont triés.

Partie 2 Définissez la fonction `addElem` qui ajoute un nouvel intervalle à un ensemble :

```
def addElem(elem: Interval, set: IntSet): IntSet = ...
```

(suite à la page suivante)

Vous pouvez admettre l'existence des deux fonctions auxilliaires suivantes :

1. `mergeable` qui prend deux intervalles et retourne vrai si et seulement si ces deux intervalles peuvent être combinés en un seul car ils sont contigus ou se chevauchent :
   ```
   def mergeable(i1: Interval, i2: Interval): Boolean
   ```
2. `merge` qui prend deux intervalles combinables (c-à-d que `mergeable` est vrai pour eux) et retourne leur combinaison :
   ```
   def merge(i1: Interval, i2: Interval): Interval
   ```

Finalement, en vous aidant de la fonction `addElem`, définissez la fonction qui calcule l'union de deux ensembles d'entiers :

```
def union(s1: IntSet, s2: IntSet): IntSet = ...
```

Exercice 3 : Images fonctionnelles (10 points)

Une image bicolore infinie peut être vue comme une fonction qui, à chaque point du plan, associe une valeur booléenne — **`true`** représentant la première couleur, **`false`** la seconde. Le type de telles images peut se définir ainsi :

```
type Image = (Point2D => Boolean)
```

où `Point2D` est le type des points du plan :

```
case class Point2D(x: Double, y: Double)
```

Etant donné cette notion d'image, on peut p.ex. définir l'image d'un carré de côté unitaire centré à l'origine :

```
val unitSquare: Image =
  { p: Point2D => abs(p.x) <= 0.5 && abs(p.y) <= 0.5 }
```

où `abs` est la fonction calculant la valeur absolue.

Affichage des images Une manière simple d'afficher des images du type `Image` ci-dessus est de les transformer en séquences de caractères qu'on imprime ensuite à l'écran. Par exemple, le caractère '`%`' peut être utilisé pour représenter les points de l'image dont la valeur est vraie, et le caractère '`.`' pour les points dont la valeur est fausse.

Votre premier but est donc de définir la fonction `imageToText` permettant de transformer une zone d'une image en une séquence de lignes. Cette fonction a la signature suivante :

```
def imageToText(i: Image, tl: Point2D, br: Point2D, w: Int, h: Int)
  : List[List[Char]] = ...
```

où `i` est l'image à transformer en texte, `tl` (pour *top left*) et `br` (pour *bottom right*) sont respectivement les coins en haut à gauche et en bas à droite de la zone de l'image à considérer, et `w` et `h` sont respectivement le nombre de colonnes et de lignes que le texte retourné doit posséder.

Par exemple, l'application suivante :

```
imageToText(unitSquare,
            Point2D(-0.7, 0.7), Point2D(0.7, -0.7),
            19, 11)
```

doit retourner une liste de 11 éléments, chacun de ces éléments étant une liste de 19 caractères représentant une ligne. Le premier caractère de la première liste est le caractère '`.`' car l'image `unitSquare` vaut **`false`** au point $(-0.7, 0.7)$. En affichant les 11×19 caractères retournés par l'application ci-dessus, on obtient comme attendu un carré centré, visible dans la moitié gauche de la figure 1.

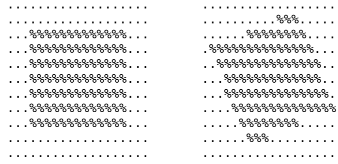

FIGURE 1 – L'image `unitSquare` de $(-0.7, 0.7)$ à $(0.7, -0.7)$ et sa version tournée de $\pi/8$

Rotation d'images Définissez une fonction qui, étant donné un angle `theta` et une image `i`, retourne une rotation de l'image autour de l'origine :

```
def rotate(theta: Double)(i: Image): Image = ...
```

Par exemple, l'application suivante :

```
rotate(math.Pi / 8.0)(unitSquare)
```

produit l'image représentée dans la moitié droite de la figure 1. Notez bien qu'un angle positif correspond à une rotation dans le sens contraire des aiguilles d'une montre.

Pour mémoire, les coordonnées du point (x', y') obtenu par une rotation d'angle θ autour de l'origine du point (x, y) sont données par :

$$\begin{aligned} x' &= x\cos(\theta) - y\sin(\theta) \\ y' &= x\sin(\theta) + y\cos(\theta) \end{aligned}$$

Vous pouvez admettre l'existence de fonctions `sin` et `cos`.

Images de Mandelbrot L'ensemble de Mandelbrot est défini comme l'ensemble des nombres complexes c tels que la séquence suivante :

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \quad n \geq 0 \end{aligned}$$

ne tend *pas* vers l'infini (en module).

Par exemple, le nombre 0 fait partie de l'ensemble de Mandelbrot car dans ce cas $\forall n, z_n = 0$. Le nombre $1 + i$ quant à lui n'en fait pas partie car la séquence tend vers l'infini en module, comme on peut s'en convaincre en calculant ses premiers termes qui sont :

$$\begin{aligned} z_0 &= 0 \\ z_1 &= z_0^2 + (1+i) = 1 + i \\ z_2 &= z_1^2 + (1+i) = 2i + 1 + i = 1 + 3i \\ z_3 &= z_2^2 + (1+i) = -7 + 7i \\ z_4 &= 1 - 97i \\ z_5 &= -9407 - 193i \\ \ldots \end{aligned}$$

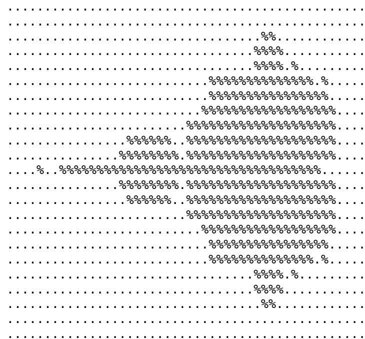

FIGURE 2 – L'image `mandelbrot` entre $(-2, 1)$ et $(0.6, -1)$

En pratique, pour déterminer si un nombre complexe c fait partie ou non de l'ensemble de Mandelbrot, on utilise une méthode approximative. Cette méthode consiste à calculer les termes successifs $z_0, z_1, \ldots$ jusqu'à ce qu'une des deux conditions suivantes soit satisfaite :

1. $|z_n| \geq 2$, auquel cas on sait que c ne fait pas partie de l'ensemble de Mandelbrot, car on peut montrer que la séquence tend forcément vers l'infini dans ce cas,
2. $n = 100$ et $|z_n| < 2$, auquel cas on admet que c fait partie de l'ensemble de Mandelbrot, 100 étant la limite arbitraire du nombre d'itérations que nous utiliserons ici.

Utilisez cette méthode approximative pour définir l'image de Mandelbrot

```
val mandelbrot: Image = ...
```

qui doit valoir vrai pour un point p de coordonnées (x, y) si et seulement si le nombre complexe $c = x + yi$ fait partie de l'ensemble de Mandelbrot. Une partie de cette image est visible sur la figure 2.

Vous pouvez vous aider de la classe des nombres complexes suivante :

```
case class Complex(re: Double, im: Double) {
  def modulus: Double = sqrt(re * re + im * im)
  def +(that: Complex): Complex =
    Complex(this.re + that.re, this.im + that.im)
  def *(that: Complex): Complex =
    Complex(this.re * that.re - this.im * that.im,
            this.im * that.re + this.re * that.im)
}
```

Si vous désirez procéder par étapes, commencez par définir le prédicat `inMandelbrotSet(c: Complex): Boolean` qui teste si un nombre complexe fait partie de l'ensemble de Mandelbrot et utilisez-le ensuite pour définir l'image elle-même.