Programmation Fonctionnelle Avancée

Juliusz Chroboczek

23 juin 2011

La durée de l'examen est de 2 heures. Les documents sont autorisés, le matériel électronique est interdit. Toutes les questions de programmation sont à traiter en Haskell ; les solutions dans d'autres langages ne seront même pas lues.

1 Fonctions strictes

Pour chacun des arguments de chacune des fonctions suivantes, dites si la fonction est stricte en cet argument. Vous n'avez pas à justifier les réponses positives ; vous devez justifier les réponses négatives en une seule équation.

```
f x = 0
g x y = y
h x y = x && y
j x y = x `seq` 42
k x = [x]
l x = x * 0
```

2 Programmation élémentaire

On rappelle l'existence des fonctions suivantes dans le module `Data.Array` :

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!) :: (Ix a) => Array a b -> a -> b
bounds :: (Ix a) => Array a b -> (a,a)
indices :: (Ix a) => Array a b -> [a]
elems :: (Ix a) => Array a b -> [b]
assocs :: (Ix a) => Array a b -> [(a,b)]
```

Le *problème des n dames* est un casse-tête consistant à placer n dames sur un échiquier $n \times n$ de façon à ce qu'aucune parmi elles n'en mette une autre en prise, i.e. qu'il n'y ait jamais deux dames sur la même colonne, la même ligne, ou la même diagonale. Par exemple, la figure 1 présente une solution au problème des quatre dames.

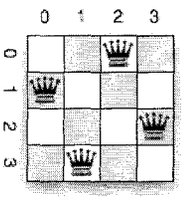

Figure 1: Une solution au problème des quatre dames

On représente un échiquier par un tableau de booléens, ou la présence d'une dame sur une case est donnée par une valeur vraie. Écrivez une fonction

```
dames :: Array (Int,Int) Bool -> Bool
```

qui retourne vrai si et seulement si le tableau a passé en paramètre représente une solution au problème des n dames, i.e. si

- a est un tableau carré ;
- a contient exactement n cases vraies, où n est la taille du tableau ;
- a ne contient jamais deux valeurs vraies sur la même ligne, la même colonne, ou la même diagonale.

3 Classes de type et monades

On définit la classe suivante :

```
class Monad m => Stash m where
  stash :: String -> m ()
```

Une instance de la classe `Stash` est une monade équipée d'une fonction `stash` qui « stocke » son argument, où la définition précise de ce que veut dire stocker dépend de l'instance.

1. Implémentez la fonction

```
stash_list :: (Show a, Stash p) => [a] -> p ()
```

qui applique `stash` aux représentations des éléments de la liste passée en paramètre, l'un après l'autre.

2. Implémentez la classe `Stash` pour le type `IO`, où `stash` affiche son paramètre sur la sortie standard.

3. On se donne maintenant le type

```
data StringWriter a = SW a String
```

3.1 Implémentez la classe `Monad` pour `StringWriter`, où `return` insère une chaîne vide, et la concaténation effectue la concaténation des chaînes.

3.2 Implémentez la classe `Stash` pour `StringWriter`, où `stash` stocke son paramètre dans la monade.

3.3 L'implémentation de >>= ci-dessus est linéaire en la taille de la chaîne déjà accumulée. Proposez *en deux phrases au plus* une modification pour la rendre constante.

4. En vous servant de l'instance ci-dessus, écrivez une fonction

```
concatenate_and_length :: [String] -> (String, Int)
```

qui retourne la concaténation des éléments de la liste passée en paramètre ainsi que la longueur de celle-ci. Par exemple,

concatenate_and_length ["tata", "tata"] = ("tatatata", 2)