Prog. Fonctionnelle Avancée: Laboratoire 9

PFA - Impression d'un arbre binaire (ordre horizontal)

Christoph Rouff Soit Rueff, Tiago Povoa Quinteiro

Contexte

> Problème de l'impression d'un arbre binaire : Si on l'imprime en le tournant de 90 degrés c'est facile, on imprime d'abord un sous-arbre, ensuite l'élément racine, ensuite l'autre sous-arbre. L'imprimer à la verticale c'est une autre histoire...

Fichiers et Modules

- `tree-print.hs` (module Main)

Exemples d'exécution

On peut voir ci-dessous des exemples d'exécution pour des types de données divers et de largeurs variables.

```
*Main> printTree t
 0
1 2

*Main> printTree t2
   1
 2   3
4 5 6 7

*Main> printTree t3
   1
 2   3
  5 6 7

*Main> printTree t4
      'a'
  'b'     'c'
'd'     'e'

*Main> printTree t5
       1
   2
 3
4
```

```
*Main> printTree t6
              1
      2               3
  4       5       6       7
8   9   10  111 12  13  14  15

*Main> printTree t7
       1
           2
             3
              4

*Main> printTree t8
              'a'
      'b'             'c'
  'd'     'e'     'f'     'g'
'h'         'k' 'l' 'm' 'n' 'o'

*Main> printTree t9
               "Albert"
     "Bruce"             "Charlie"
"David"   "Eric"    "Fabien"  "Greg"

*Main> printTree t10
     [1,2,3]
[1,2,3,4] [2,3,5,2]
```

Implémentation

Implémentation de l'arbre binaire

Au départ, on avait imaginé notre arbre comme ceci

```haskell
data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a) deriving
(Show)
```

Soit une feuille, soit un node. Par la suite, nous avons réalisé que ça revenait à un node avec deux enfants vides.

```haskell
data BinaryTree a = Nil | Node a (BinaryTree a) (BinaryTree a) deriving (Show)
```

Comme on peut le voir au-dessus, on l'a du coup simplement retiré.

Exemple de déclaration

Voici un exemple de déclaration d'arbre en haskell.

```haskell
t = Node 0 (Node 1 Nil Nil) (Node 2 Nil Nil)
t2 = Node 1 (Node 2 (Node 4 Nil Nil) (Node 5 Nil Nil)) (Node 3 (Node 6 Nil Nil)
(Node 7 Nil Nil))
```

On peut ainsi construire un arbre en spécifiant la valeur qu'il contient et ses deux enfants.

Première approche

Avant de démarrer l'impression de l'arbre à la vertical, on a essayé de le faire à l'horizontale. Ainsi, on a pu se faire une idée du processus avec un problème beaucoup plus simple à résoudre.

```
toStringH Nil = "Empty Tree\n"
toStringH bt = ts 0 bt
  where
    ts l (Leaf a) = spaces l a
    ts l Nil = ""
    ts l (Node a left right) = (ts (l + 1) right) ++ (spaces l a) ++ (ts (l + 1)
left)

printTreeHorizontal t = do
  putStr (toStringH t)
```

Note: à cette étape, nous utilisions encore `leaf`.

Le fait d'avoir fait cette approche avant l'autre nous a permis d'identifier certains besoins dans le calcul d'un arbre à la verticale.

Impression de l'arbre vertical

Première version

Calcul de la hauteur

Afin de pouvoir calculer correctement le décalage horizontale de chaque étage de l'arbre, il nous faut savoir sa hauteur. En effet, on sait que la toute dernière ligne en bas sera collée au bord. Et chaque étage consécutif au-dessus de cette dernière sera décalé en conséquence.

```
height Nil = 0
height (Node _ left right) =
  let
    hl = height left
    hr = height right
  in
    (if hl > hr then hl else hr) + 1
```

Version améliorée

Informations supplémentaires

Jusqu'alors, on avait calculé la hauteur et on déduisait le décalage. Le problème était d'avoir un espacement fixe. Il ne s'adaptait pas à la taille réelle des éléments, ni tenait compte de la possibilité de différentes largeurs d'éléments.

Exemple:

- `1` vs `12` vs `123`

```
treeInfo Nil = (0, 1)
treeInfo (Node a left right) =
  let
    size = length (show a)
    (hl, sl) = treeInfo left
    (hr, sr) = treeInfo right
  in
    ((maximum [hl, hr]) + 1, (maximum [size, sl, sr]))
```

Marge à gauche

```
leftPadding height level unit
  | height == level = 0
  | otherwise = (2^(height - level - 1)) * (unit + 1) - (div unit 2) - 1
```

Grâce au calcul précédent, on peut donc déduire la marge à gauche à appliquer pour décaler correctement les éléments et donner la typique forme triangulaire à l'arbre.

Espace inter-éléments

```
interSpaces height level unit =
  if level == height then
    unit
  else
    2 * (interSpaces height (level + 1) unit) + 1
```

Par rapport à la hauteur global de l'arbre, sa hauteur à lui et l'espacement unitaire minimale (unit) de l'arbre: Calcule le nombre d'espacements à ajouter après un élément au sein de l'arbre.

Impression

La fonction que l'utilisateur finale va appeler est `printTree`.

Dans celle-ci, on appelle `treeInfo` vu dans la section précédente.

```
printTree tree = do
  let (h, s) = treeInfo tree
  printLevels h s tree
```

À ce moment là, on a toutes les informations nécessaires pour la partie suivante.

Afin de tacler ce problème, on a besoin d'une approche itérative: concrètement il s'agit d'imprimer l'arbre ligne par ligne (ou niveau par niveau).

Donc `printLevels` va appeler la fonction interne `levelsPrintLoop`.

```
printLevels height maxSize tree = do levelsPrintLoop 1 tree
  where
    unit = if maxSize `mod` 2 == 0 then maxSize + 1 else maxSize
    levelsPrintLoop i tree = do
      putStr (spaces (leftPadding height i unit))     -- print left padding
      putStr (levelToString height unit i tree)       -- print line
      if i == height then do
        putStr "\n"
      else do
        putStr "\n"
        levelsPrintLoop (succ i) tree
```

Le calcul de `unit` prend la taille maximale que peut avoir un élément dans l'arbre.

Si nécessaire, on ajoute +1 pour que cet espacement unitaire soit impaire et que l'arbre puisse donc être centré correctement.

La fonction suivante, `levelToString` prépare la string d'une ligne à imprimer.

```
{-
Returns a given level as a string by computing addapted spaces offsets
for the elements.
If some elements are missing due to an empty parent (Nil at a higher level), it
computes
the right amount of spaces this missing parent causes at this level.
Ex : If there is a Nil at level 2 (element missing) then there will be 4 missing
elements at level 4, so we need to add spaces for 4 elements.
-}
levelToString height unit level tree = helper level tree
  where
    padding s = drop s (spaces (interSpaces height level unit))
    -- missing element on given level
    helper 1 Nil = ' ' : (padding 0)
    -- missing element on a higher level
    helper i Nil = repeat' (' ' : (padding 0)) (2^(level - i))
    helper 1 (Node a left right) =
      let
        stringA = show a
      in
        stringA ++ (padding ((length stringA) - 1))
    helper i (Node a left right) =
      helper (i - 1) left ++ helper (i - 1) right
```

Une grande difficultée a été d'adapter ce code en tenant compte que certains emplacements sont vides. Prenez l'exemple d'un arbre fortement désequilibré. Tous les noeuds manquants d'un côté vont générer des trous qui vont rendre difficile à positionner l'arbre.

Dans cette fonction, on peut injecter le bon nombre d'espaces pour pallier à ce problème.

Utils

Génère une chaîne d'espaces de longueur **n**.

```
spaces 0 = ""
spaces n = ' ' : spaces (n - 1)
```

repeat' permet de répeter une string **n** fois.

```
repeat' s 0 = ""
repeat' s n = s ++ (repeat' s (n - 1))
```

Conclusion

Imprimer un arbre à l'horizontale (couché) est trivial. Le faire à la verticale est beaucoup plus compliqué qu'on peut le penser. Surtout s'il doit s'adapter à tout type de données.

On est partit initialement sur une résolution récursive, mais on a pu constater que ce n'était pas nécessairement adapté. On a donc dû approcher le problème de manière séquentielle.

Une amélioration qu'on pourrait encore apporter serait d'ajouter des barres pour lier ensemble les éléments de l'arbre. Plus joli, mais semble être relativement compliqué.