Principes de Programmation
Seconde Phase du Projet de Programmation

21 février 2019

Objectifs : Rentrer dans le code de votre camarade, modifier sans tout réécrire, faire un bibliothèque centré sur l'utilisation et non l'algorithme (abstraction).

Il va maintenant falloir découpler la bibliothèque de la phase 1 en deux bibliothèques :

- `Ensemble`, qui est une version abstraite des ABRI définis précédemment.
- `Dictionnaire`, qui est une version plus complète des ABRI.

De plus, il vous faudra rendre les types polymorphiques afin d'accepter, en particulier, des caractères comme clefs.

Remarque : les fichiers doivent avoir comme nom `Ensemble.hs` et `Dictionnaire.hs`, ils doivent être mis dans une archive `zip` ou `tar.gz`.

Polymorphisme Les ABRI que l'on a fait ne sont définis que sur les entiers, mais on peut les définir sur n'importe quel type `a` qui dispose d'un ordre discret (avec prédécesseur et successeur) afin de pouvoir définir des segments. Dans ce genre de cas, on ne contraint pas le type algébrique, mais les fonctions utilisées. Ainsi, la signature de nos fonctions change et, par exemple, l'insertion prend le type :

```
inserer :: (Ord a, Enum a) => ABRI a -> a -> ABRI a
```

Remarquez que l'on utilise les deux classes Ord et Enum pour représenter cet ordre discret. Cela ne demande pas beaucoup d'autre changements, uniquement l'utilisation de `pred :: Enum a => a->a` et `succ :: Enum a => a->a` au lieux de `-1` et `+1`, qui existent déjà en Haskell et sont définis, en particulier, pour les entiers et les caractères.

Ensemble Une implémentation possible des ensembles consiste à utiliser un ABRI dont les clés sont les éléments de l'ensemble. Il s'agit d'une abstraction car l'utilisateur n'a pas à savoir qu'il y a derrière un ABRI. De plus, on procède au renomage de certaines fonctionnalités.

Dictionnaire Les dictionnaires ont une structure plus complexe : ils associent des valeurs à des clés. Une implémentation possible est d'utiliser des ABRI dont les noeuds ont, en plus d'un segment de clés, une valeur. Il vous faudra donc modifier la définition du type algébrique des ABRI. De plus, il faut faire attention à ne fusionner les segments que s'ils réfèrent à une même valeur.

Vous allez devoir :

1. Rendre polymorphe le module de la phase 1.
2. Le copier en deux exemplaires, renommez le premier `Ensemble` et le second `Dictionnaire`.
3. Changer les noms des fonctions de `Ensemble` pour coller au canevas.
4. Ajouter les fonctions d'union et d'intersection comme donné sur le canevas.
5. Changez le type algébrique de dictionnaire et les fonctions associées. [1]
6. Ajoutez une fonction `foldSet` comme décrit dans le canevas. [2]
7. Changez les noms des fonctions de `Dictionnaire` quand nécessaire.
8. Ajoutez la fonction `rechercherParCle` qui prends un dictionnaire, une clé et rend la valeur associée lorsqu'elle existe (dans un `Just`) et rend un `Nothing` si elle n'existe pas.
9. Ajoutez la fonction `mapDico` qui prend une fonction sur les valeurs et un arbre, puis applique la fonctions sur toutes les valeurs de l'arbre. [3]
10. Ajoutez la fonction `filtrer` qui prend une fonction `f :: cont -> Bool` sur les valeurs vers les bouléens et un arbre, puis qui supprime les valeurs `x` telles que `(f x)` est faux.

1 Bonus 1 :

L'insertion dans le dictionnaire d'une clé déjà utilisée mais avec une valeur différente n'est pas définit, vous pouviez rendre ce que vous voulez. Pour le bonus, il s'agit d'écrire les version sécurisés (avec maybe) est insertions.

2 Bonus 2 (ouvert) :

L'utilisation de `Ord` et `Enum` ne décrit pas vraiment un ordre discret correctement, et ce pour plusieurs raisons :

1. On suppose que l'on essait pas d'insérer un couple (cle,val) dans un dictionnaire qui associe une valeur diférente de val à cle.
2. (Bonus) Vous pouvez essayer d'implémenter la type-class `Foltable` et fixer `foldSet` = `fold`
3. (Bonus) Vous pouvez essayer d'implémenter la type-class `Functor` et `fixermapDico` = `fmap`. On ne demande pas de recompresser l'arbre s'il y a des projections de valeurs.

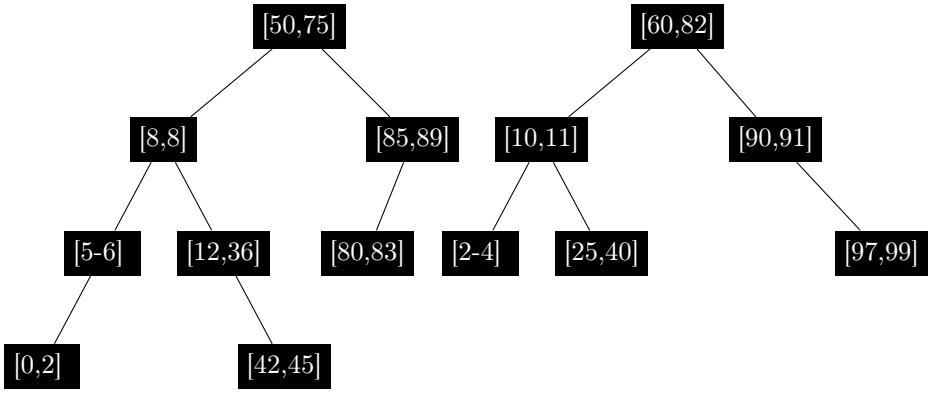
— Le comportement aux bornes est mal défini, ainsi `pred '\NUL'` rend une erreur car `'\NUL'` est le premier caractère.
— On peut avoir un prédécesseur et un successeur sans pour autant s'injecter dans les entiers, comme par exemple `(a,Int)` avec `pred (x,y) = (x,y-1)` et `succ (x,y) = (x,y+1)`.
— Si on autorise des segments ouverts et des segments fermés, on peut encore diminuer la contrainte et ajouter les flottants. Attention tout de même : dans les entiers, les segments $[\![2, 42]\!]$ et $[\![2, 43[\![$ sont les mêmes, il faut donc une contrainte de type cohérente.

Pour pallier à ces problèmes, il faudrait changer la type-class de nos fonctions et utiliser autre chose que `Enum` qui n'est pas tout à fait adapté. Il existe assez peu d'alternative dans la librairie standard de Haskell (on pourra tout de même regarder ce que l'on peut faire avec la classe `Bounded`).

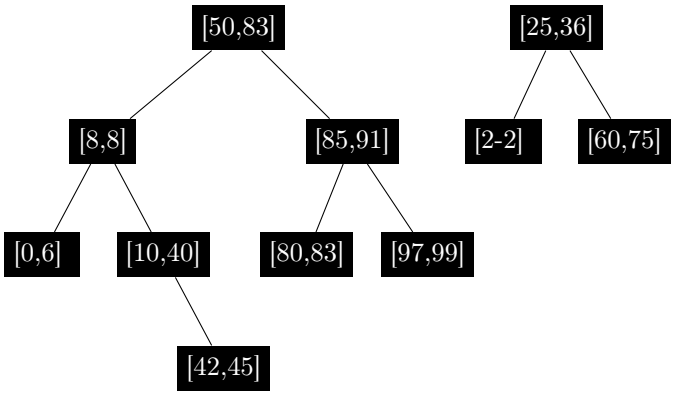
Dans la vrais vie, il est rarement recommandé de créer vos propre classes et il vaut mieux rester avec cette version imparfaite à moins d'avoir un cas vraiment important qui n'est pas implémentable sans généralisation. Mais c'est un bon exercice de se poser la question "si j'en avais vraiment besoin, comment je ferais ?"

Vous avez bien sure le droit de changer de canevas, mais dans ce cas, faites deux version : une sans changement et une avec ; la seconde doit avoir le nom `DictionnaireBonus`.

Exemple : Sur les ensembles `ensemble1` et `ensemble2` definit comme :



l'union `union ensemble1 ensemble2` et l'intersection `intersection ensemble1 ensemble2` doivent renvoyer :



remarque : la position des noeuds dans l'arbre n'est pas importante, faites au plus rapide, puis essayez d'équilibrer si vous pouvez.