TRAVAUX PRATIQUES XVII

Arbres Rouge-Noirs

L'objectif de ce TP est d'obtenir une implémentation persistante[1] des arbres rouge-noir en OCaml : test d'appartenance, insertion. L'ajout de la suppression donne la version étoilée du TP.

A Introduction

A.1 Fonctionnement du TP

A.1.i : Todos

L'implémentation intégrale des ARN est un peu longue. Aussi avec ce TP vous est fourni un squelette : `arn.ml` . Il contient certaines fonctions pré-implémentées et d'autres contenant des `failwith "TODO"` . Votre travail est de remplacer ces `failwith "TODO"` par du code correct (et éventuellement de changer le filtrage par motif associé).

Attention, vous croiserez aussi d'autres `failwith` : ceux-ci ne sont pas des trous à compléter, ce sont des messages d'erreur (souvent associés à des cas impossibles) qui sont là pour aider au débuggage.

Par exemple, la fonction `etiquette` contient `| Nil -> failwith "etiquette Nil"` . Ce n'est pas un trou à compléter, c'est une erreur pour vous aider à débuguer : vous avez voulu accéder à l'étiquette de Nil qui n'en a pas.

A.1.ii : Interface

En plus du squelette, vous trouverez aussi `arn_doc.mli` . C'est un fichier interface (comme les headers en C) qui contient la documentation des fonctions.

Pour en créer un format plus agréable à lire, je recommande l'utilisation de `ocamldoc` : cette commande terminal transforme un `mli` en page html. Pour l'utiliser, ouvrez un terminal et allez dans le dossier du `mli` puis lancez : `ocamldoc arn_doc.mli -html` . Vous obtiendrez beaucoup de fichiers html, dont `Arn_doc.html` : c'est ce dernier qui vous intéresse ! Vous n'avez plus qu'à l'ouvrir avec votre navigateur, par exemple `firefox Arn_doc.html` .

Si vous voulez que ces html soient générés ailleurs, regardez l'option `-d` de `ocamldoc` .

Enfin, vous trouverez `ex.ml` : c'est un fichier qui contient des exemples bien choisis d'ARN. Vous devriez l'utiliser pour tester vos fonctions. Vous devriez aussi ne pas vous limiter à ces exemples, et en faire d'autres qui correspondent à vos besoins.

A.1.iii : Let's go

Votre mission, et vous l'acceptez, est de compléter les trous de `arn.ml` afin de respecter la spécification indiquée dans `arn_doc.mli` . En particulier, à la fin les fonctions de Recherche et d'Insertion doivent fonctionner[2].

1. Aussi appelé « structure de données fonctionnelle », c'est à dire que les insertions/suppressions ne modifient pas en place un ARN, mais en renvoient un nouveau.
2. Attention, il ne s'agit pas simplement de réussir à Rechercher et Insérer, mais aussi et surtout de le faire dans des Rouge-Noirs.

A.2 Définitions

On utilisera la définition suivante d'Arbres Rouges Noirs :

Définition 1.
Un ARN arbre binaire de recherche, aux étiquettes deux à deux distinctes (pas de doublon). On ajoute en outre une couleur à chaque noeud : Rouge ou Noir. Il doit vérifier les conditions suivantes :

(0) C'est un Arbre Binaire de Recherche, c'est à dire que le parcourt infixe lit les étiquettes par ordre strictement croissant.

(1) Aucun noeud Rouge n'a de fils Rouge.

(2) Tous les chemins de la racine à un Nil contiennent le même nombre de noeuds noirs (sans compter la racine).

(3) La racine est Noire.

Remarque : la propriété (3) empêche de définir les ARN par induction. Pour désambuiguiser, dans ce sujet on appelle :

- *arbre rouge-noir correct* un arbre vérifiant toutes les conditions.
- *sous-arbre rouge-noir correct* un arbre vérifiant toutes les conditions sauf éventuellement la (3).
- *sous-arbre rouge-noir presque correct en Rouge* un arbre vérifiant les conditions (0), (2), (et éventuellement (3)). En outre, s'il ne vérifie pas (1), alors cette dernière est rompue une seule fois (il existe un unique enchainement de noeuds Rouges) **et** cette rupture à la racine (la racine est Rouge, et a un enfant qui est rouge).

Dans ce sujet, on utilise le type OCaml suivant pour représenter les couleurs.

```
type couleur = Rouge
             | Noir
```

À l'aide de ce type `Couleur` , on définit de la manière suivante un arbre Rouge-Noir.

```
type 'a arn = Nil
            | Node of couleur * 'a arn * 'a * 'a arn
```

Vous noterez que c'est un type immuable : nos fonctions ne devront pas *modifier* un arbre, mais bien en renvoyer un nouveau sur lequel a été appliquée l'opération voulue. Afin de simplifier ses phrases, cet énoncé mélangera parfois ces deux notions.

Pour compiler et tester vos fonctions, vous avez deux options :

- Travailler en mode compilé. Dans ce cas, faites débuter le fichier `ex.ml` devrait débuter par `open Arn` . Cela donne accès aux fonctions de `arn.ml` sans avoir à ajouter le préfixe `Arn.` : ainsi, si `arn.ml` définit `etiquette` , alors `ex.ml` pourra utiliser cette fonction en l'appelant simplement `etiquette` (et non `Arn.etiquette`).
  Pour compiler vos deux fichiers ensembles : `ocamlopt arn.ml ex.ml`
- Travailler en mode interpreté. En ouvrant `utop` depuis le dossier de vos fichiers, `#use "file.ml"` permet de compiler `file.ml` et de charger ses fonctions dans l'interpréteur, puis de même pour `ex.ml` .
  L'interpréteur peut-être plus pratique dans un premier temps : l'affichage des arbres est fort commode !

Un dernier conseil pour la route :

FAITES. DES. DESSINS !!

B Mise en jambe

1. Représentez l'arbre `arn_ex` de `ex.ml` sur un schéma.

On s'intéresse à la partie « Accesseurs ». Elle a deux sous-parties[3] : « Propriétés d'un noeud » et « Propriétés d'un arbre ».

Cette partie vise à vous familiariser avec le type.

2. **a.** Complétez les codes de la sous-partie « Propriétés d'un noeud ».
   b. Complétez les codes de la sous-partie « Propriétés d'un arbre ». C'est notamment ici que se trouvent les fonctions de recherche. Pour `hauteur_noire`, on utilise la définition du cours : nombre d'arcs *entrants* dans un noeud Noir sur un chemin de la racine à Nil.
   c. Testez vos fonctions.

C Insertion

Le principe de l'insertion est celui vu dans le cours : on crée une nouvelle feuille rouge avec la clé à insérer, puis on corrige les problèmes en remontant jusqu'à la racine.

Comme dans la méthode vue en cours, on ne violera jamais la condition d'équilibre noir. Le seul problème potentiel sera un nœud rouge avec un enfant rouge, et la résolution de ce problème sera la responsabilité du parent (nécessairement noir) du premier noeud rouge.

Les fonctions de recolorations et de rotations serviront dans la partie étoilée, vous pouvez les ignorer pour l'instant.

C.1 Cas d'insertion

L'objectif est maintenant de coder la fonction `corrige_rouge`. Tout d'abord, à l'aide d'un papier et d'un crayon, retrouvez les 4 cas de correction R-R de l'insertion vus en cours.

3. Complétez les trous de la fonction `corrige_rouge`.
4. Testez. Testez. Testez. Testez.

C.2 Insertion en elle-même

Pour insérer, on va procéder de la manière suivante : on descend récursivement jusqu'à trouver où placer la nouvelle feuille Rouge contenant l'élément à insérer. Ensuite, on remonte récursivement en appliquant à chaque étape `corrige_rouge` : cela permet de faire remonter l'éventuel enchainement de Rouge jusqu'à la racine initiale !

La fonction `insere_aux` effectue cela.

5. **a.** Completez le cas de base de `insere_aux`.
   b. Lisez le reste du code. Comprenez-vous pourquoi il réalise bien la descente / remontée expliquée ci-dessus ? Pour vous en assurer, expliquez-la à votre peluche comme si elle avait 5 ans.

Il reste un cas que nous n'avons pas traité : si l'enchainement R-R a lieu à la racine.

6. Complétez la fonction `insere`. Vous devrez vous assurer d'obtenir un arbre rouge-noir correct.
7. Testez. Testez. Testez. Testez.
8. **a.** Écrire une fonction `arn_of_list : 'a list -> 'a arn` qui prend en argument un tableau sans doublons et renvoie un arn contenant tous ses éléments.
   b. Écrire une fonction `sort_by_arn` qui trie un tableau en utilisant le parcours de l'arn associé au tableau.
   c. *Bonus : ajoutez ces fonctions et leur documentation au* `.mli`

3. Cette notion de parties/sous-parties se voit très bien dans le html produit par `ocamldoc`.

Figure XVII.1 – Un pingouin très fier de vous.