TP11: Arbres binaires de recherche

MP2I Lycée Pierre de Fermat

Prélude: gestions d'exceptions

OCaml possède un mécanisme appelé les **exceptions**, qui permet de manipuler des erreurs dans le flot normal du programme. Les exceptions permettent d'écrire du code comme:

```
Essayer:
    x ← fonctionPouvantFaireUneErreur();
    retourner x
Si une erreur est levée:
    si l'erreur était ErreurA alors
        retourner 0
    si l'erreur était ErreurB alors
        retourner −1
```

En OCaml, la syntaxe est proche du match-with, et utilise les mots clés **try** et **with**:

```
try E with
  | exception1 -> valeur1
  ...
  | exceptionk -> valeurk
```

Pour interpréter une telle expression:

1. Évaluer E en une valeur v. Si aucune erreur n'est rencontrée, alors l'expression a comme valeur v

2. Si une erreur est rencontrée, matcher l'erreur avec exception1, ..., exceptionk, et renvoyer la valeur correspondante. Si aucun match n'est trouvé, l'erreur n'est pas rattrapée.

Par exemple, lorsque l'on divise par 0 en OCaml, cela lève une exception appelée `Division_by_zero`. On peut utiliser le mécanisme de rattrapage d'exception pour écrire une fonction de division qui, en cas de division par 0, rattrape l'erreur et ne la laisse pas interrompre le programme:

```
(* renvoie x / y si y != 0, et renvoie None si y = 0 *)
let division_safe (x: int) (y: int) : int option =
  try Some (x / y) with
  | Division_by_zero -> None

let test () =
  assert (division_safe 10 2 = Some 5);
  assert (division_safe 7 0 = None)
```

Deuxième exemple: la fonction `failwith` lève une exception `Failure of string`. On peut alors la rattraper comme suit:

```
(* max_liste l renvoie l'élément maximal de l. Lève une erreur Failure si la liste est vide. *)
let rec max_liste (l: 'a list) : 'a =
  match l with
  | [] -> failwith "Liste vide"
  | [x] -> x
  | x :: q -> max x (max_liste q)

(* max_opt l renvoie l'élément maximal de l. Renvoie None si la liste est vide. *)
let max_opt (l: 'a list) : 'a option =
  try Some (max_liste l) with
  | Failure _ -> None
```

Q1. Lorsqu'une assertion échoue, elle lève une exception `Assert_failure of string`. Écrire une fonction `valider: (unit -> unit)-> bool` telle que si `test: unit -> unit` est une fonction des tests avec assertions, `valider test` renvoie `true` si les tests réussissent, `false` sinon.

Arbres binaires de recherche

On implémente dans cette partie des ABR dont les informations sont stockées sur les noeuds, que l'on utilisera pour implémenter la structure d'ensemble. On notera les arbres en notation infixe, on propose donc la notation suivante:

```
type 'a arbre =
  | V
  | N of 'a abr * 'a * 'a abr (* (gauche, etiquette, droite) *)

(* recherche x a renvoie un booléen indiquant si x est dans a.
   Précondition: a est un ABR *)
let recherche (x: 'a) (a: 'a arbre) : bool = ...

(* ajoute x a renvoie un ABR contenant x et les éléments de a. Si x est déjà dans a, ajoute x a renvoie a.
   Précondition: a est un ABR*)
let ajoute (x: 'a) (a: 'a arbre) : 'a arbre = ...

(* supprime x a renvoie un ABR contenant les éléments de a sauf x.
   Si x n'est pas un élément de a, lève une erreur Failure. *)
let supprime (x: 'a) (a: 'a arbre) : 'a arbre = ...
```

Q2. Implémenter les fonctions `recherche` et `ajoute`, en $\mathcal{O}(h)$ avec h la hauteur de l'arbre en entrée.

Q3. Écrire une fonction `extraire_max: 'a arbre -> 'a * 'a arbre` telle que `extraire_max a` renvoie le couple (m, a') avec m l'étiquette maximale de a, et a' est un arbre obtenu en supprimant m dans a. En utilisant cette fonction, implémenter la fonction `supprime`.

Q4. On définit par récurrence la suite d'arbres $(A_n)_{n\in\mathbb{N}}$:

- $A_0 = V$
- $\forall n \in \mathbb{N}, A_{n+1} = \textbf{ajoute}(n+1, A_n)$

Renseignez-vous sur la fonction `Sys.time`. En l'utilisant, écrivez une fonction `time_a: int -> float` qui calcule le temps nécessaire pour créer l'arbre $A(n)$. Déterminer expérimentalement le temps que demande la création de $A(n)$ (sous la forme d'un $\mathcal{O}$), et expliquer le résultat obtenu.

Arbres rouge-noir

On considère maintenant des arbres dont les informations sont stockés aux feuilles. Le but de cette partie est d'implémenter les arbres rouge-noir vus en cours. On utilisera les types suivants:

```
type couleur = Rouge | Noir

type 'a noeud_arn =
  | Feuille of 'a
  | Noeud of couleur * 'a * 'a noeud_arn * 'a noeud_arn

type 'a arn = 'a noeud_arn option (* None représente l'arbre vide *)
```

Lorsqu'on implémentera une fonction sur les ARN, on implémentera d'abord une fonction sur le type `'a noeud_arn`, qui sera récursive et implémentera la vraie logique de la fonction, puis la fonction principale sur le type `'a arn` qui rajoutera une surcouche permettant de gérer le cas où l'arbre est vide. Par exemple, pour calculer le nombre d'éléments contenus dans un arn (i.e. le nombre de feuilles):

```
let feuilles (a: 'a arn) : int =
  let rec feuilles_noeud (aa: 'a noeud_arn) : int = match aa with
    | F x -> 1
    | N(c, x, g, d) -> feuilles_noeud g + feuilles_noeud d
  in match a with
  | None -> 0
  | Some r -> feuilles_noeud r
```

Commençons par écrire des fonctions permettant de vérifier qu'un arbre satisfait bien les conditions des ARN.

Q5. Écrire une fonction qui calcule la couleur de la racine d'un `'a noeud_arn`.

Q6. Écrire une fonction qui détermine si un `'a noeud_arn` contient deux noeuds rouges successifs.

Nous allons maintenant écrire une fonction calculant la hauteur noire d'un `'a noeud_arn`. Cette fonction devra en même temps vérifier que tous les chemins de la racine aux feuilles ont le même nombre de noeuds noirs, et lèvera une erreur si ce n'est pas le cas.

En OCaml, on peut **étendre** le type des exception, et rajouter une expression faite maison:

```
exception Erreur_hauteur_noire
```

Cette ligne rajoute une exception, et dans la suite du code on peut écrire `raise Erreur_hauteur_noire` pour lever l'exception rajoutée.

Q7. Écrire une fonction `hauteur_noire: 'a noeud_arn -> int` calculant la hauteur noire d'un ARN, et levant une exception `Erreur_hauteur_noire` s'il y a une incohérence, i.e. si deux feuilles sont à des distances noires de la racine différentes.

Q8. En utilisant les fonctions précédentes, écrire une fonction `arn_valide` permettant de vérifier qu'un arbre de type `'a arn` est bien un ARN. Testez la sur de nombreux exemples pour vérifier que vous couvrez bien toutes les possibilités.

On implémente maintenant l'insertion. On rappelle que pour faire une insertion dans un ARN, on effectue une insertion classique d'ABR, en colorant le nouveau noeud en rouge. Puis, on corrige les éventuelles anomalies. Pour cela, on utilise une fonction auxiliaire qui permet de transformer un arbre tel que l'un des enfants ainsi qu'un de ses enfants sont rouges en un ARN valide mais dont la racine peut être rouge. Référez vous au cours pour les quatre cas possibles.

Le schéma d'insertion sera donc:

- Pour insérer dans une feuille, on crée un nouveau noeud permettant de séparer les deux feuilles du nouvel arbre
- Sinon, on insère récursivement dans le bon sous-arbre, puis on corrige l'arbre obtenu.

Q9. Implémenter une fonction `correctionARN: 'a noeud_arn -> 'a noeud_arn` qui corrige l'anomalie d'un ARN et le transforme en ARN relaxé. Cete fonction ne sera pas récursive.

Q10. Implémenter une fonction `insertARNrelax: 'a -> 'a noeud_arn -> 'a noeud_arn` qui insère dans un ARN, et renvoie un arbre dont la racine est éventuellement rouge.

Q11. En déduire une fonction `insertionARN` permettant d'insérer un élément dans un arbre rouge-noir. L'arbre résultant devra être un ARN, vérifiez le dans vos tests en utilisant la fonction de vérification que vous avez codé.

On définit la suite d'arbres $(B_n)_{n\in\mathbb{N}}$ comme suit:

- $B_0 = F(0)$
- $B_{n+1} = \textbf{insertionARN}(n+1, B_n)$ pour $n \in \mathbb{N}$

Q12. Écrivez une fonction permettant de calculer la suite $(B_n)_{n\in\mathbb{N}}$. Dessinez B_{14}, vérifiez à la main que c'est un ARN et donnez sa hauteur noire.

Q13. Écrivez une batterie de tests permettant de vérifier que les arbres B_n sont bien des ARN, et que la hauteur de B_n est inférieure à $2\log_2(2n+1)$.

Q14. Comparez les temps de création des B_n à celui des A_n, la suite d'arbres binaires de recherche introduite à la question précédente. Quelle est la complexité temporelle de la création de B_n ?

On s'intéresse maintenant à l'utilisation des ARN pour implémenter une fonction de tri. Pour trier une liste L, l'idée est d'insérer chaque élément de L dans un ARN, puis de calculer la liste des éléments de l'ARN dans l'ordre croissant.

Q15. Écrire une fonction `listerARN` permettant de lister dans l'ordre croissant tous les éléments d'un ARN. N'oubliez pas que seules les feuilles encodent l'information !

Q16. (**Bonus**) Faire la fonction précédente en $\mathcal{O}(n)$, donc sans utiliser la concaténation.

Q17. En déduire une fonction `tri_arn: 'a list -> 'a list` permettant de trier une liste en utilisant un ARN. Quelle est sa complexité ?

Cette application des ARN motive leur utilisation pour implémenter des ensembles ou des dictionnaires. Nous avons vu plus tôt dans l'année que les tables de hachage permettent des complexité en $\mathcal{O}(1)$ pour les opérations d'insertion, de suppression et de recherche. Cependant, l'information dans une table de hachage ne suit aucune structure par rapport aux clés. Dans un ABR, l'information est stockée en suivant l'ordre des clés. Ceci permet de faire certaines requêtes sur les ABR de manière très efficace par rapport aux tables de hachage.

Q18. Écrire une fonction permettant d'obtenir l'étiquette minimale d'un ARN.

Q19. Écrire une fonction `query_range: 'a -> 'a -> 'a arn -> 'a list` telle que `query_range a b t` renvoie la liste des éléments de t compris entre a et b.

Q20. (**Bonus**) Implémenter la suppression dans les ARN.