BE PF2 2023–2024 — Documents autorisés — durée 3h00

Les sections 1 et 2 sont indépendantes l'une de l'autre mais la première constitue une bonne mise en jambes.

1 Représentations binaires d'entiers

Dans cet exercice, vous développerez une implémentation possible des entiers naturels représentés en base 2. Le fichier `Nat_intf.ml` contient la spécification de cette implémentation dans la signature `S`.

L'implémentation se fait dans le module `Nat_bin.Bin`. Le principe est de représenter les entiers binaires comme des listes de chiffres (`0` ou `1`) avec le **poids faible à gauche**. La liste non-vide `[ b0; ...; bk ]` représente l'entier $\sum_{i=0}^{k} b_i \times 2^i$. Par exemple, la liste `[ 1; 0; 0; 1; 1 ]` représente l'entier 25.

Les contraintes suivantes doivent être respectées :

— le nombre 0 (à ne pas confondre avec le chiffre `0`) est représenté par la liste vide `[]`

— le chiffre le plus à droite d'une liste doit être différent de `0`.

Vous devrez implémenter les fonctions suivantes :

```
(* ajoute 1 à un entier *)
val inc : t -> t
(* décrémente un entier de 1 ; si l'entier représentait 0, renvoyer ce même nombre (ne pas lever d'exception) *)
val dec : t -> t
(* additionne deux entiers binaires *)
val add : t -> t -> t
(* renvoie l'int correspondant au naturel en paramètre *)
val to_int : t -> int
```

■ **Exercice 1:** Compléter le module `Nat` dans le fichier `nat.ml`.

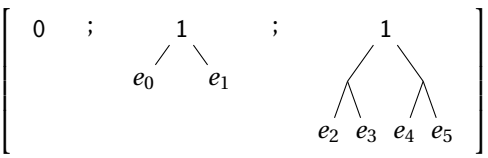
2 Listes à accès direct

La liste à accès direct (*random-access list*, RAL) est une structure de données purement fonctionnelle qui combine les opérations fondamentales des listes (p. ex. ajouter un élément en tête de liste) et des tableaux (p. ex. obtenir l'élément stocké à un indice donné) avec une bonne complexité algorithmique.

2.1 Structure de données à base d'arbres binaires

La structure de données RAL peut intuitivement être vue comme une liste fondée sur la représentation *binaire* des entiers vue en § 1. Comme dans celle-ci, chaque cellule dispose d'un marqueur de présence (correspondant à 1) ou d'absence (correspondant à 0) ; et le poids d'une cellule *double* à chaque décalage vers la droite. Autrement dit, la cellule i (en débutant à 0), si elle contient des éléments (de type `'a`), en contient 2^i ; sinon elle en contient 0. Comment assurer la présence de 2^i éléments ? En stockant ces éléments dans un arbre binaire parfait (avec stockage aux feuilles) de profondeur i. Pour rappel, un arbre binaire parfait est un arbre binaire équilibré qui comprend le même nombre d'élément stockés dans son sous-arbre gauche et son sous-arbre droit. In fine, les RAL sont organisées comme des listes de longueur logarithmique d'arbres de hauteur logarithmique : c'est ce qui est à l'origine de leurs bonnes performances.

Voici par exemple une RAL contenant 6 (`011` en binaire) éléments (p. ex. l'élément e_2 est à l'indice 2). Le `0` ou `1` sur la ligne du haut indique, comme expliqué plus haut, si un arbre est présent ou pas dans cette case.



Le module `Ral_trees.M` comprend un squelette d'implémentation de cette structure. Il définit d'abord un type pour les arbres binaires avec stockage aux feuilles. Puis, comme en § 1, on définit une notion de "nombre" binaire constitué de `0` et de `1`, à ceci prêt que le `1` indique le stockage d'un arbre :

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a digit = Zero | One of 'a tree
type 'a t = 'a digit list
```

La liste exemple est codée ainsi (en écrivant des entiers dans les cases) :

```
[Zero; One (Node (Leaf 0, Leaf 1)); One (Node (Node (Leaf 2, Leaf 3), Node (Leaf 4, Leaf 5)))]
```

On remarque en passant que rien n'impose au niveau du type qu'un arbre soit parfait. De même, il faudra garantir que le poids d'une case double à chaque décalage à droite et que la liste ne finit pas par un `0`. Enfin, l'ordre des éléments est de gauche à droite, dans la liste mais aussi aux feuilles des arbres. Tout cela sera assuré dans l'implémentation des opérations.

Voici la liste des fonctions dans la spécification `Ral_inft.S` qu'il faudra implémenter :

```
(* met un élément en tete de liste *)
val cons : 'a -> 'a t -> 'a t
(* renvoie l'élément en tête de liste ; lève Empty en cas d'usage sur une liste vide *)
val head : 'a t -> 'a
(* renvoie la liste amputée de son élément de tête ; lève Empty en cas d'usage sur une liste vide *)
val tail : 'a t -> 'a t
(* `get n l` renvoie le n-ième élément de la liste `l` en comptant àpartir de 0 ;
   si l'élement n'est pas présent, lever l'exception Not_found *)
val get : int -> 'a t -> 'a
```

■ **Exercice 2:** Implémenter la fonction `cons`.

■ **Exercice 3:** Implémenter les fonction `head` et `tail`.
Indication : on peut à cet effet définir une fonction `uncons` qui renvoie la tête et la queue d'une RAL.

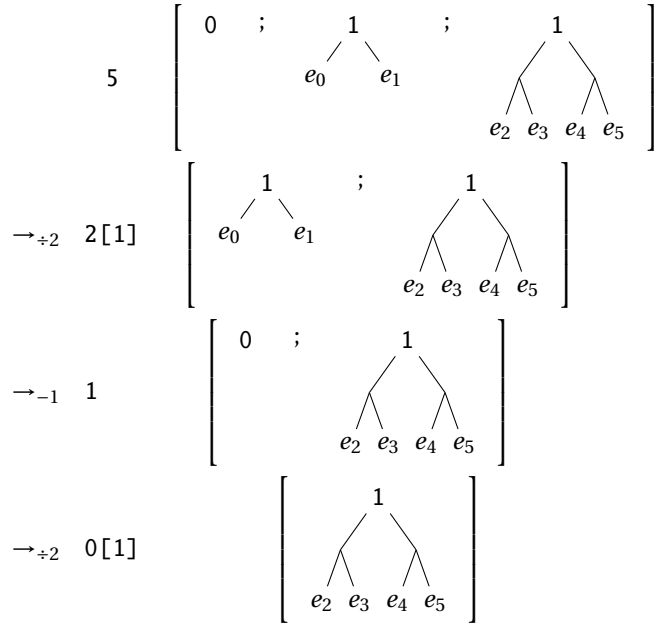
■ **Exercice 4:** Implémenter la fonction `get`.

Indication : plusieurs approches sont possibles. On peut par exemple parcourir la liste en calculant, pour chaque cellule de la liste, la taille possible pour l'arbre qui peut y être stocké (laquelle double à chaque fois), et avancer en mettant à jour l'indice recherché (si la case est un `1` / `One`). Une fois que l'indice est plus petit que la taille de l'arbre dans la cellule où on se situe, on a trouvé le bon arbre...

Une deuxième approche **(pas obligatoire ici mais obligatoire pour l'exercice 7)**, élégante, fonctionne sans calculer la taille mais plutôt en exploitant directement le lien avec la représentation binaire. Supposons qu'on cherche l'élément d'indice 5 (soit 101) dans la liste plus haut. L'idée principale est la suivante :

— dans le cas général, on regarde la tête de liste et si c'est un `0`, on décale l'indice *et* la liste vers la gauche, et on recommence. Décaler l'indice vers la gauche revient à le diviser par 2. Toutefois, il ne faut pas oublier le reste de la division. Si celui-ci est nul, le décalage n'a pas "oublié d'information" et l'élément recherché sera à gauche dans l'arbre retourné par l'appel récursif. En revanche, si le reste est 1, il ne faut pas oublier cette quantité supplémentaire, ce qui correspond à aller vers la droite dans l'arbre. *Note : l'opérateur OCaml pour calculer le reste de* `x` *modulo 2 s'écrit* `x mod 2`.
— si la tête de liste est un 1, on peut soustraire 1 dans l'indice et remplacer la tête de liste par un `0`.
— quand l'indice est devenu `0` et que la case courante est un 1, on a trouvé le bon arbre. Il faut le retourner et les appels plus haut dans la pile d'exécution parcourront l'arbre à gauche ou à droite en fonction des restes de divisions discutés ci-dessus.

La figure suivante illustre le déplacement dans la liste tel que décrit ci-dessus. Le nombre entre crochets est le reste de la division. Ici, deux restes à 1 indiquent qu'il faudra aller deux fois dans le sous-arbre droit retourné à la fin du déplacement, ce qui renverra e_5.



2.2 Structure de données à base d'arbres *nécessairement* parfaits

Un défaut de la structure précédente est, qu'en principe, rien n'assure que les arbres stockés dans la liste sont effectivement parfaits. En cours 5 et TD 5, nous avions vu un type permettant d'assurer cet invariant au moyen du **type non-uniforme** suivant : `type 'a perfect_tree = | Empty | Node of 'a * ('a * 'a) perfect_tree`.

Nous nous inspirons de ce type pour définir un **nouveau type non-uniforme** pour implémenter la RAL avec arbres nécessairement parfaits (cf. `Ral_pairs.M`) :

```
type 'a t =
  | Nil                      (* RAL vide *)
  | Zero of ('a * 'a) t      (* cellule vide suivie d'une queue *)
  | One of 'a * ('a * 'a) t  (* cellule non-vide suivie d'une queue *)
```

La liste exemple est ainsi codée par : `Zero (One ((0, 1), One (((2, 3), (4, 5)), Nil)))` (en écrivant des entiers dans les cases).

Compléter `Ral_pairs.M` pour qu'il soit conforme à la spécification `Ral_intf.S` :

■ **Exercice 5:** Implémenter la fonction `cons`.

■ **Exercice 6:** Implémenter les fonction `head` et `tail`.
Indication : on peut à cet effet définir une fonction `uncons` qui renvoie la tête et la queue d'une RAL.

■ **Exercice 7:** Implémenter la fonction `get` en utilisant la "deuxième approche" décrite dans l'indication de l'exercice 4.