

Implémentation d'un type abstrait "ensemble d'entiers naturels"

3h

Avec documents

Année 2023-2024 — Session 1

Préambule

- Rappel : VS Code signale parfois des erreurs qui n'en sont pas, il ne faut se fier qu'au retour du compilateur (`dune runtest` ou `dune utop`).
- Pour tester dans `utop` : **open Be**; puis ouvrir les modules que vous voulez tester.
- Le code rendu doit **impérativement compiler**. Si une partie ne compile pas, mettez-la en commentaires, ça peut donner lieu à des points.
- Interdiction de modifier les signatures (nom + type) indiquées dans les contrats.
- Même si la non utilisation d'itérateur sera pénalisée, des points seront accordés si les fonctions sont écrites correctement sans itérateur.
- Les documents autorisés sont : vos notes de cours, TD et TP, le site d'OCaml et Moodle. Vous êtes autorisé à utiliser toutes les fonctions de la librairie standard, en particulier le module `List` : <https://ocaml.org/api/List.html>.
- Utilisez le script `genererRendu.sh` pour générer un fichier `<votre login>.tar` compression d'un répertoire `<votre login>` contenant les fichiers `base.ml`, `liste.ml`, `arbre.ml`, `conversion.ml`, `natset.ml` et les fichiers `dune`. Avant de déposer le tar, vérifiez qu'il contient bien les fichiers souhaités.
- Le barème est donné à titre indicatif et pourra être modifié.

Introduction

L'objectif de ce bureau d'étude est la réalisation d'un module de gestion d'ensembles contenant uniquement des entiers naturels. Les opérations mises en jeu sont le test d'appartenance, l'ajout et le retrait. Il est naturel d'essayer d'obtenir une implantation plus efficace que l'implantation "naïve" à base de listes, dans laquelle la plupart des opérations ont une complexité linéaire en la taille de l'ensemble. Nous proposons ici de concrétiser cette démarche à l'aide d'une structure de données arborescente permettant l'implantation d'opérations de complexité logarithmique en la taille de l'ensemble.

1 Principe

Les entiers seront décomposés dans une base `base` et l'ensemble sera représenté par un arbre dont le type OCaml est le suivant :

```
type arbre = Noeud of bool * arbre list | Vide
```

Chaque fils devra avoir **exactement** `base` nombre de fils.

Chaque noeud d'un arbre ne représente un entier que s'il contient le booléen `true` et dans ce cas, la décomposition de cet entier en base `base` est lue depuis ce noeud jusqu'à la racine de l'arbre, du poids faible vers le poids fort. La *i*-ième branche représente le bit *i*. Par convention, l'entier 0 est représenté

par une liste vide de bits, par conséquent, s'il appartient à un ensemble, il sera nécessairement stocké à la racine de l'arbre correspondant. Par convention également, et dans un souci d'économie et de simplicité, seuls les bits significatifs seront représentés dans les arbres. Ceci implique que le premier bit est toujours non nul (sauf pour l'entier 0) et que la racine de l'arbre correspondant à un ensemble non vide admet toujours **Vide** comme premier fils.

Enfin, toujours dans un souci de simplicité, aucune feuille d'un arbre ne doit être de la forme **Noeud(false, [Vide ;...; Vide])**, ce qui signifie que toutes les suites de bits présentes dans les branches d'un arbre représentent un préfixe (au sens large) du code d'au moins un entier. Aucune branche n'est donc inutile.

On veillera à ce que les différentes opérations implantées préservent bien cette propriété de normalité. Par hypothèse, tous les arbres passés en paramètre seront également normaux.

Exemple de représentation en utilisant la base 2

A titre d'exemple, nous donnons la représentation graphique simplifiée de l'ensemble $E = \{0, 1, 5, 6, 7\}$ ainsi que sa représentation sous forme de constructeurs.

```
Noeud(true , [Vide;
  Noeud(true , [Noeud(false, [Vide;
    Noeud(true , [Vide; Vide ])] ) ;
    Noeud(false, [Noeud(true , [Vide; Vide ];
      Noeud(true , [Vide; Vide ]))]]))
```

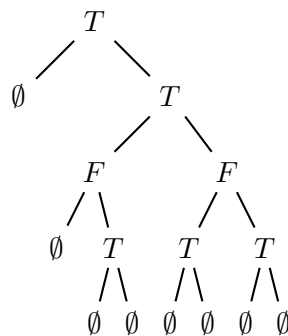
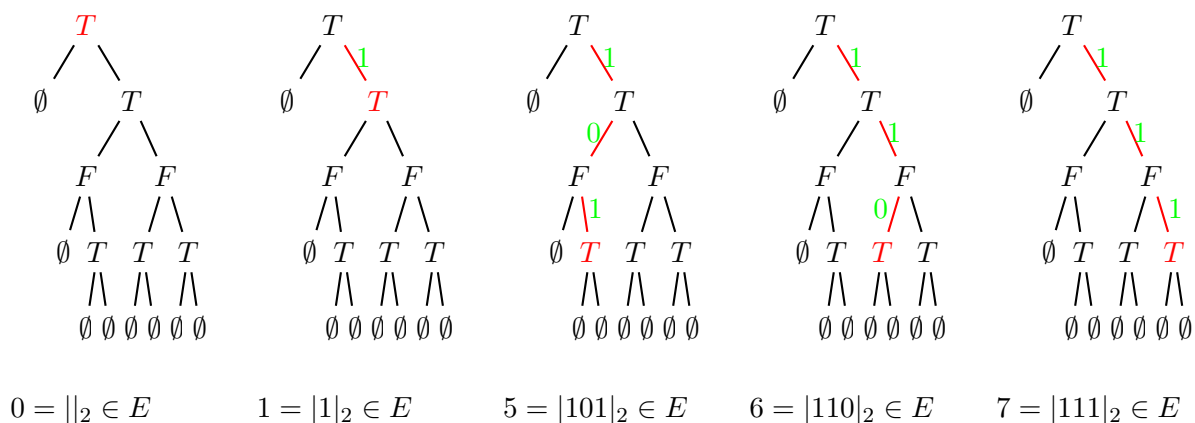


FIGURE 1 – Un exemple d'ensemble basé sur la décomposition en base 2 des entiers



Décomposition en base 5

A titre d'exemple, nous donnons la représentation graphique simplifiée de l'ensemble $E = \{0, 1, 7, 14, 36\}$ ainsi que sa représentation sous forme de constructeurs.

```

Noeud (true ,
[Vide;
  Noeud (true ,
    [Vide; Vide;
      Noeud (true ,
        [Vide; Noeud (true , [Vide; Vide; Vide; Vide; Vide]); Vide; Vide; Vide]);
        Vide; Vide]);
  Noeud (false ,
    [Vide; Vide; Vide; Vide; Noeud (true , [Vide; Vide; Vide; Vide; Vide]);
    Vide;
    Vide])

```

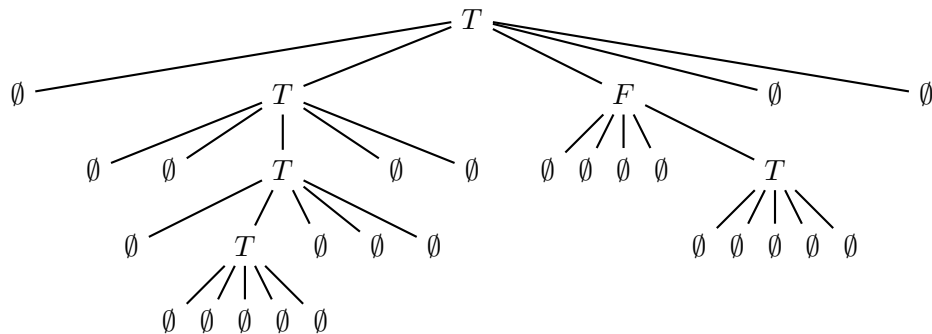
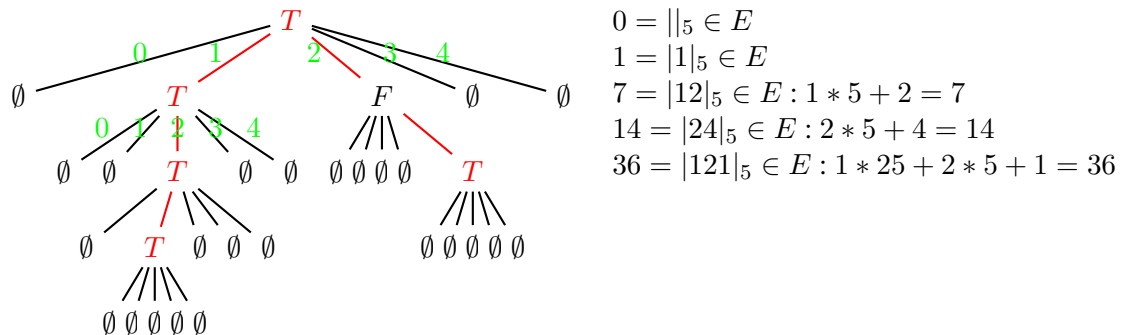


FIGURE 2 – Un exemple d'ensemble basé sur la décomposition en base 5 des entiers



2 Travail demandé

Exercice 1 : base.ml – 1 point

Écrire deux modules **Base2** et **Base5** conforme à **Base** et correspondant à une base 2 et une base 5.

Indication : Il est normal que vous trouviez cela simple, car en effet, cela l'est.

Exercice 2 : `liste.ml` – 6,5 points

1. Écrire le contrat et les tests unitaires de la fonction `l_taille` qui construit une liste constituée de n (passé en paramètre) fois le même élément e (passé en paramètre).
2. Écrire la fonction `l_taille`.
3. Définir l'exception `ArgumentInvalide`.
4. Écrire la fonction `get` qui renvoie le i -ième élément d'une liste. Pensez à dé-commenter les tests pour tester la fonction.
5. Écrire la fonction `set` qui positionne le i -ième élément d'une liste. Pensez à dé-commenter les tests pour tester la fonction.

Indication : vous pouvez utiliser l'itérateur `List.mapi` qui est une variante de `List.map` (voir la documentation OCaml).

Utilisation des tests fournis

Pour les trois exercices suivants, les tests sont donnés dans des modules (ou foncteurs) commentés à différents niveaux. Il faudra dé-commenter au fur et à mesure.

Par exemple pour l'exercice 3 :

```
1 (*
2 module TestConversion2 = struct
3   module Conversion2 = Conversion (Base2)
4   open Conversion2
5
6   let %test_unit _ =
7     print_string "=== Tests du module Conversion en Base 2 ===\n"
8
9   (* decompose *)
10  (*
11    let %test _ = decompose 0 = []
12    ...
13    *)
14  (* recompose *)
15  (*
16    let %test _ = recompose [] = 0
17    ...
18    *)
19 end
20
21 module TestConversion5 = struct
22   module Conversion5 = Conversion (Base5)
23   open Conversion5
24
25   let %test_unit _ =
26     print_string "=== Tests du module Conversion en Base 5 ===\n"
27
28   (* decompose *)
29   (*
30     let %test _ = decompose 0 = []
31     ...
32     *)
33   (* recompose *)
34   (*
```

```

35  let %test _ = recompose [] = 0
36  ...
37  *)
38  end
39  *)

```

- Pour la question 3.1 (écriture du foncteur) supprimer `(* et *)` correspondant aux lignes 1 et 39 (évidemment les numéros de lignes ne seront pas celles-ci dans votre code).
- Pour la question 3.2 (écriture de la fonction `decompose`) supprimer `(* et *)` correspondant aux lignes 10, 13, 29 et 32.
- Pour la question 3.3 (écriture de la fonction `recompose`) supprimer `(* et *)` correspondant aux lignes 15, 18, 34 et 37.

Exercice 3 : `conversion.ml` – 4 points

Les contrats des fonctions sont donnés dans le fichier `conversion_contrats.txt`. Reportez les contrats dans le fichier `ml`.

1. Écrire un foncteur `Conversion` paramétré par une `Base`. Dé-commenter les modules de tests.
2. Écrire, dans le foncteur `Conversion`, la fonction `decompose` : `int -> int list` qui réalise une décomposition d'un entier. La base de la décomposition est donnée par le module paramétrant le foncteur. Le bit de poids fort sera le premier élément de la liste. Dé-commenter les tests de `decompose`.

Indications :

- Nous choisissons que `decompose 0 = []`.
- `decompose 14 = [1;1;1;0]` en base 2.
 - $14 \text{ div } 2 : q=7 \text{ et } r=0$
 - $7 \text{ div } 2 : q=3 \text{ et } r=1$
 - $3 \text{ div } 2 : q=1 \text{ et } r=1$
 - $1 \text{ div } 2 : q=0 \text{ et } r=1$
- 3. Écrire, dans le foncteur `Conversion`, la fonction `recompose` : `int list -> int` qui reconstruit un entier à partir de sa décomposition. La base de la décomposition est donnée par le module paramétrant le foncteur. Dé-commenter les tests de `recompose`.

Exercice 4 : `arbre.ml` – 8,5 points

Les contrats des fonctions sont donnés dans le fichier `arbre_contrats_type.txt`. Reportez les contrats dans le fichier `ml`.

Indication : Il est conseillé d'utiliser les fonctions écrites dans l'exercice 2.

1. Écrire un foncteur `Arbre` paramétré par une `Base` et contenant le type `arbre` explicité précédemment. Dé-commenter les modules de tests.
2. Écrire, dans le foncteur `Arbre`, la fonction `conforme` : `arbre -> bool` qui teste si un arbre est conforme à la base qui paramètre le module. Pour être conforme, tous les nœuds doivent avoir le même nombre de fils, qui correspond à la base de décomposition. Dé-commenter les tests de `conforme`.
3. Écrire, dans le foncteur `Arbre`, la fonction `appartient` : `int list -> arbre -> bool` qui teste l'appartenance d'un entier, décomposé dans la base, à un ensemble. Dé-commenter les tests de `appartient`.
4. Écrire, dans le foncteur `Arbre`, la fonction `ajouter` : `int list -> arbre -> arbre` qui ajoute un entier, décomposé dans la base, à un ensemble. Dé-commenter les tests de `ajouter`.

5. Écrire, dans le foncteur **Arbre**, la fonction **retirer** : **int list** -> **arbre** -> **arbre** qui retire un entier, décomposé dans la base, d'un ensemble et doit renvoyer un ensemble normalisé. Dé-commenter les tests de **retirer**.

Indication : Pour illustrer le problème de la normalisation, la figure 3 trace les différentes étapes pour normaliser le résultat de l'opération $\{0, 1, 5, 6, 7\} \setminus \{5\} = \{0, 1, 6, 7\}$, en base 2. L'arbre de gauche est similaire à celui de la figure 2, la feuille codant l'entier 5 contenant maintenant **false**. L'arbre de droite est le résultat final attendu.

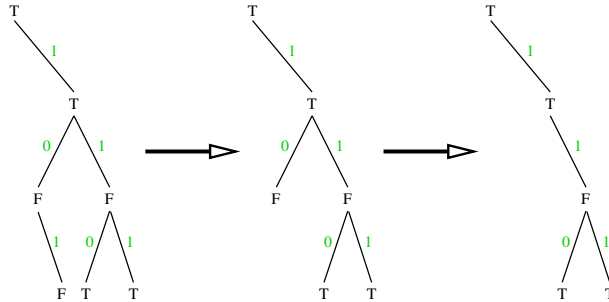


FIGURE 3 – Normalisation de $\{0, 1, 5, 6, 7\} \setminus \{5\} = \{0, 1, 6, 7\}$

Exercice 5 (BONUS) : **natset.ml** – 3,5 points

Les contrats des fonctions sont donnés dans le fichier `natset_contrats.txt`. Reportez les contrats dans le fichier `ml`.

1. Écrire un foncteur **NatSet** paramétré par une **Base**. Dé-commenter le foncteur et les modules de tests.
2. Ajouter à ce foncteur deux modules **C** et **A**, respectivement l'instanciation du foncteur **Conversion** et du foncteur **Arbre** avec la **Base** paramétrant le foncteur **NatSet**.
3. Définir le type **natset** représentant un ensemble dans la base donnée.
4. Définir **vide** : **natset** représentant l'ensemble vide.
5. Écrire la fonction **ajouter** : **int** -> **natset** -> **natset** qui ajoute un entier à un ensemble. Dé-commenter les tests de **ajouter**.
6. Écrire la fonction **appartient** : **int** -> **natset** -> **bool** qui teste l'appartenance d'un entier à un ensemble. Dé-commenter les tests de **appartient**.
7. Écrire la fonction **retirer** : **int** -> **natset** -> **natset** qui retire un élément d'un ensemble et doit renvoyer un ensemble normalisé. Dé-commenter les tests de **retirer**.