Autour des entiers naturels

1 Parties de $\mathbb{N}$

1.1 introduction

On s'intéresse ici aux parties de l'ensemble des entiers naturels $\mathbb{N}$. Il n'existe pas de solutions informatiques pour représenter une partie quelconque de $\mathbb{N}$. Nous allons cependant voir différentes manières de représenter certaines de ces parties, afin de pouvoir effectuer des opérations sur celles-ci.

On se permettra d'utiliser les fonctions vues en cours sur les listes, telles que `List.mem` qui retourne un booléen indiquant la présence d'un élément dans une liste.

1.2 Parties finies

On s'intéresse dans un premier temps aux parties *finies* de $\mathbb{N}$. On représentera, dans un premier temps, une partie finie de $\mathbb{N}$ par une liste des entiers [1] la constituant, *sans doublon*. Par exemple, `[ 0; 1; 2; 3 ]` ou `[ 2; 0; 3; 1 ]` représentent tous deux l'ensemble des entiers strictement inférieurs à 4. La représentation n'est donc pas unique.

1. Écrire une fonction `est_vide` de signature `int list -> bool` (ou bien `'a list -> bool`) prenant en argument une liste définissant une partie finie de $\mathbb{N}$ et retournant un booléen indiquant (en temps constant $O(1)$) si elle est vide.

2. Écrire une fonction `union` de signature `int list -> int list -> int list` (ou bien `'a list -> 'a list -> 'a list`) prenant en entrée deux parties finies de $\mathbb{N}$ et retournant une liste d'entiers représentant leur union (et ne contenant donc pas de doublon). On pourra, par exemple, ajouter à la première liste les éléments de la seconde liste qui ne figurent pas dans la première. Quelle est la complexité, dans le pire des cas, de cette fonction ?

3. Écrire de même deux fonctions `inter` et `diff`, de mêmes signatures, réalisant les opérations ensemblistes d'intersection et de différence (on retirera les éléments de la seconde partie à la première, dans ce second cas).

4. Montrer que deux parties $\mathcal{P}_1$ et $\mathcal{P}_2$ sont disjointes si et seulement si le résultat d'une opération ensembliste bien choisie sur ces deux parties donne une partie vide.

5. En déduire une fonction `disj` de signature `int list -> int list -> bool` indiquant par un booléen si deux parties de $\mathbb{N}$, fournies en argument à la fonction, sont disjointes.

6. De façon similaire, proposer une fonction `inclus` de signature `int list -> int list -> bool` prenant en argument deux listes décrivant des parties finies de $\mathbb{N}$, et retournant un booléen indiquant si la première est incluse dans la seconde.

7. Enfin, proposer une fonction egaux de signature `int list -> int list -> bool` prenant en argument deux listes décrivant des parties finies de $\mathbb{N}$, et retournant un booléen indiquant si elles sont égales. Cette fois encore, on écrira le test en réfléchissant à la meilleure façon d'utiliser des opérations ensemblistes déjà définies.

1.3 Parties finies et cofinies

L'ensemble des parties finies de $\mathbb{N}$ n'est pas stable par la complémentation (le complément $\overline{A}$ d'une partie A étant l'ensemble des entiers de $\mathbb{N}$ n'appartenant pas à A).

On définit une partie *cofinie* de $\mathbb{N}$ comme une partie de $\mathbb{N}$ dont le complément est une partie finie de $\mathbb{N}$. On peut montrer que l'ensemble $\mathcal{W}$ des parties finies ou cofinies de $\mathbb{N}$ est stable pour les opérations d'union, d'intersection, de différence et de complémentation. C'est d'ailleurs le plus petit ensemble des parties de $\mathbb{N}$ qui vérifie ces propriétés.

On remarquera que $\mathcal{W}$ ne correspond pas à l'ensemble des parties de $\mathbb{N}$: l'ensemble des entiers pairs, ou l'ensemble des nombre premiers, par exemple, ne sont ni des parties finies, ni des parties cofinies !

On définit le type

```
type partie = Finie of int list | Cofinie of int list;;
```

« `Finie [ 2; 3; 5 ]` » désigne simplement la partie finie de $\mathbb{N}$ constituée des entiers 2, 3 et 5. « `Cofinie [ 2; 3; 5 ]` » doit quant à elle s'entendre comme le complémentaire dans $\mathbb{N}$ de l'ensemble $\{2,3;5\}$, c'est-à-dire la partie de $\mathbb{N}$ constituée de 0, de 1, de 4 et des entiers supérieurs ou égaux à 6.

8. Proposer une fonction `est_vide_W` de signature `partie -> bool` qui prend en argument un élément de $\mathcal{W}$ et retournant un booléen indiquant si cette partie de $\mathbb{N}$ est vide.

9. Écrire une fonction `compl_W` de signature `partie -> partie` qui prend en argument un élément de $\mathcal{W}$ et retourne son complémentaire.

10. Écrire deux fonctions Caml appelées `union_W` et `inter_W` de signature `partie -> partie -> partie` qui prennent en argument deux éléments de $\mathcal{W}$ et retournent respectivement leur union et leur intersection.

11. Érire deux fonction `disj_W` et `inclus_W` de signature `partie -> partie -> bool` indiquant par un booléen si deux éléments de $\mathcal{W}$, fournis en argument à la fonction, sont disjoints dans le cas de `disj_W`, et si le premier est inclus dans le second pour `inclus_W`.

12. Écrire une fonction `egaux_W` de signature `partie -> partie -> bool` qui indique

1. On oubliera, le temps de ce TP, qu'il existe des entiers que OCaml ne peut représenter, car trop grands.

si les deux arguments correspondent au même élément de $\mathcal{W}$.

13. Écrire une fonction `complementaires_W` de signature `partie -> partie -> bool` qui indique si les deux arguments sont complémentaires (leur intersection est disjointe, leur union est $\mathbb{N}$).

1.4 Ensemble des parties de $\mathbb{N}$

Il existe des parties de $\mathbb{N}$ qui ne sont ni finies, ni cofinies, comme les nombres pairs, les nombres premiers, etc. On peut cependant définir des parties élaborées de $\mathbb{N}$ via un prédicat (une propriété qui serait vraie pour les entiers qui la constituent, et fausse pour les autres).

Par exemple, l'ensemble des entiers pairs peut être associé à la fonction « `fun x -> (x mod 2 = 0)` » : les entiers pairs correspondent exactement aux entiers qui, soumis à cette fonction, donnent un résultat « `true` ».

On peut donc définir de très nombreuses parties de $\mathbb{N}$ par un objet de signature `int -> bool`. On notera $\mathcal{U}$ l'ensemble des parties de $\mathbb{N}$ qui peuvent être définies par une fonction f, c'est-à-dire les ensembles de la forme $\{i \in \mathbb{N} \mid f(i) = \texttt{true}\}$.

14. Comment créer une fonction `union_U` qui réalise l'union de deux parties de $\mathbb{N}$ définies ainsi ? Sa signature doit être `(int -> bool) -> (int -> bool) -> int -> bool` ou bien `('a -> bool) -> ('a -> bool) -> 'a -> bool`.

15. Définir de même `inter_U`, `diff_U` et `compl_U`.

Si l'on omet le fait que OCaml ne permet pas de représenter tous les entiers, il n'est en revanche plus possible de tester si un ensemble défini par une fonction est vide (il faudrait vérifier tous les entiers de $\mathbb{N}$ un à un, ce qui prendrait un temps infini !). De même, il est impossible d'écrire des fonctions vérifiant l'égalité, l'inclusion ou la complémentarité de deux parties.

16. Écrire une fonction `convert` de signature `int list -> int -> bool` qui prend en argument une liste d'entiers représentant une partie finie de $\mathbb{N}$ et retourne un objet de signature `int -> bool` désignant cette même partie finie de $\mathbb{N}$.

17. Écrire de même une fonction `convert_W` de signature `partie -> int -> bool` qui prend en argument un partie finie ou cofinie de $\mathbb{N}$ et retourne un objet de signature `int -> bool` désignant cette même partie de $\mathbb{N}$.

1.5 Opérateur universel

Cette partie est facultative, en fonction du temps dont vous disposez.

On définit la fonction

```
let foo f1 f2 = function x -> not ((f1 x)&&(f2 x));;
```

Cette fonction est universelle, on peut écrire chacune des fonctions précédentes à partir de `foo`.

18. Réécrire `union_U`, `inter_U`, `diff_U` et `compl_U` *uniquement à partir de* `foo` (on s'interdit donc l'usage de tout autre opérateur : les fonctions ne doivent contenir que des appels à `foo`, les noms des arguments, et des parenthèses).

2 Algèbre de Peano

2.1 Introduction

Une façon de décrire l'ensemble des entiers naturels a été proposée par G. Peano. Elle repose sur cinq axiomes :

- l'élément appelé « *zéro* » et noté 0 est un entier naturel ;
- tout entier naturel n a un unique successeur, noté Sn qui est un entier naturel ;
- aucun entier naturel n'a 0 pour successeur ;
- deux entiers naturels ayant le même successeur sont égaux ;
- si un ensemble d'entiers naturels contient 0 et contient le successeur de chacun de ses éléments, alors cet ensemble est $\mathbb{N}$.

Pour représenter l'ensemble des entiers naturels, on peut donc utiliser le type suivant :

```
type pint = Zero | S of pint;;
```

Ainsi, zéro correspond à l'élément « `Zero` », trois correspond à « `S (S (S Zero))` ».

19. Proposer une fonction `int_of_pint` de signature `pint -> int` prenant en argument un entier en représentation de Peano et retournant sa représentation usuelle.

20. En déduire une fonction `print_pint` affichant, dans la représentation usuelle, un entier dans la représentation de Peano (on supposera que l'entier en question ne dépasse pas les capacités de représentation des entiers de OCaml).

21. Proposer de même une fonction `pint_of_int` de signature `int -> pint` effectuant la conversion inverse.

2.2 Opérations élémentaires

On définit l'addition de deux entiers en utilisant les propriétés suivantes de l'addition :

$$\begin{cases} x+0 = x \\ x+(y+1) = (x+1)+y \end{cases}$$

22. En déduire une fonction `add` de signature `pint -> pint -> pint` calculant la somme de deux entiers en représentation de Peano (évidemment, sans passer par une conversion en entiers naturels).

Pour la multiplication de deux entiers, on pourra par exemple écrire :

$$\begin{cases} x \times 0 = 0 \\ x \times (y+1) = (x \times y) + x \end{cases}$$

23. En déduire une fonction `mul` de signature `pint -> pint -> pint` calculant le produit de deux entiers en représentation de Peano.

24. Quelle est la complexité de cette opération de multiplication ?

On n'oubliera pas d'effectuer quelques tests pour vérifier chacune des fonctions écrites !

2.3 Une multiplication différente

On propose ici une approche différente pour la multiplication.

25. Proposer une fonction `is_even` de signature `pint -> bool` retournant un booléen indiquant si l'argument est pair.

26. Proposer une fonction `div2` de signature `pint -> pint` prenant en argument un entier n en représentation de Peano et retournant $\lfloor n/2 \rfloor$.

27. Proposer une fonction `mul2` de signature `pint -> pint` prenant en argument un entier n en représentation de Peano et retournant $2n$.

On défini alors la multiplication de la sorte :

$$\begin{cases} x \times 0 = 0 \\ x \times y = (2 \times x) \times \lfloor y/2 \rfloor & \text{si } y \text{ est pair} \\ x \times y = (2 \times x) \times \lfloor y/2 \rfloor + x & \text{si } y \text{ est impair} \end{cases}$$

28. En déduire une fonction `mul_bis` de signature `pint -> pint -> pint` déterminant le produit de deux entiers.

29. A-t-on obtenu une meilleure complexité ?

2.4 Aller plus loin

30. Proposer une fonction `fact` de signature `pint -> pint` prenant en argument un entier n en représentation de Peano et retournant sa factorielle.

31. Écrire une fonction `divmod` de signature `pint -> pint -> pint * pint` prenant en argument un entier a et un entier b non nul en représentation de peano et retournant les entiers q et r tels que $a = q \times b + r$.

32. En déduire une fonction `pgcd` de signature `pint -> pint -> pint` calculant le PGCD de deux entiers en notation de Peano.