DS1 d'informatique MPSI
4 mars 2011

Devoir surveillé

Durée : 2 heures

Aucun document n'est autorisé. **On pourra définir des fonctions non demandées explicitement, si cela facilite la programmation. Inutile de prouver la correction et la terminaison des fonctions écrites, sauf si on le demande explicitement. Les fautes de syntaxe seront sanctionnées.**

Cases atteintes par un cavalier en p coups

Dans ce problème, on considère les déplacements d'un cavalier sur un échiquier. L'échiquier sera carré, de taille $n \times n$ (où n est un entier supérieur ou égal à 3), que l'on modélisera mathématiquement par l'ensemble $\mathcal{E}_n = [\![0, n-1]\!]^2$ (où $[\![0, n-1]\!] = \{i \in \mathbb{N}, 0 \leqslant i \leqslant n-1\}$), et dont les éléments seront appelés *cases*. Comme d'habitude, il y a autant de cases blanches que de cases noires, placées en damiers. Plus précisément, la case (i, j) est blanche si et seulement si $i + j$ est pair.

Un cavalier situé sur l'échiquier en position (i, j) peut se déplacer (en un coup) sur les cases de l'échiquier distantes d'une case dans une direction (horizontale ou verticale) et de deux dans l'autre, et seulement sur ces cases. On note en particulier que la couleur de la case sur laquelle se trouve le cavalier change à chaque coup.

Définition : dans la suite, on appellera *voisine* d'une case (i, j) toute case de l'échiquier qu'un cavalier placé en (i, j) peut atteindre en un coup.

On définit donc la fonction suivante, associant à une case (i_0, j_0) la liste des huit cases qu'un cavalier est susceptible d'atteindre en partant de (i_0, j_0).

```
# let deplace_temp (i0, j0) =
        [(i0 - 2, j0 + 1); (i0 - 2, j0 - 1);
        (i0 + 2, j0 + 1); (i0 + 2, j0 - 1);
        (i0 - 1, j0 + 2); (i0 + 1, j0 + 2);
        (i0 - 1, j0 - 2); (i0 + 1, j0 - 2)];;
deplace_temp : int * int -> (int * int) list = <fun>
```

On prendra cependant garde au fait que parmi ces huit cases possibles, toutes ne sont pas nécessairement sur l'échiquier : si on part par exemple de la case $(0, 0)$, le cavalier ne peut atteindre que les cases $(1, 2)$ et $(2, 1)$.

Problème principal de ce sujet : on s'intéresse aux cases qu'un cavalier peut atteindre en partant d'une case (i, j) de l'échiquier, et en au plus p coups : par exemple, pour $p = 0$, la seule case est bien sûr (i, j), et, pour $p = 1$, ce sont (i, j) et ses voisines (au sens de la définition ci-dessus).

Pour traiter ce problème, nous allons choisir une modélisation fonctionnelle et un traitement récursif dans la partie B, une modélisation par matrices [1] dans les deux parties suivantes, le traitement mélangera programmations impérative et récursive dans la partie C, et sera purement impératif dans la partie D.

Le sujet se conclut par une partie où l'on se pose d'autres problèmes, similaires.

Nous utiliserons la fonction prédéfinie make_matrix de type int -> int -> 'a -> 'a vect vect, telle que make_matrix n p a soit la matrice de taille (n, p), dont tous les termes valent a. Par exemple :

```
# make_matrix 2 2 0;;
- : int vect vect = [|[|0; 0|]; [|0; 0|]|]
```

Pour accéder au terme en position (i, j) d'une matrice t, on écrira t.(i).(j) :

```
# let exemple = [|[|1; 2; 3|]; [|4; 5; 6|]|];;
exemple : int vect vect = [|[|1; 2; 3|]; [|4; 5; 6|]|]

# exemple.(0).(1);;
- : int = 2

# exemple.(1).(2);;
- : int = 6
```

1. vues comme de simples tableaux de nombres ou de booléens

En poursuivant cet exemple, on peut facilement obtenir le nombre de lignes et de colonnes d'une matrice :

```
(* nombre de lignes d'une matrice *)

# vect_length exemple;;
- : int = 2

(* Nombre de colonnes d'une matrice *)

# vect_length exemple.(0);;
- : int = 3
```

Partie A – Préliminaires

A.1 (question non essentielle pour la suite)

a Définir une fonction print_bool de type bool -> unit qui affiche le caractère V (resp. F) si l'argument vaut true (resp. false).

b Définir une fonction affiche_bool : bool vect vect -> unit, prenant en argument un tableau de booléens, et affichant le tableau correspondant dont les termes sont les caractères V et F. On aura par exemple :

```
# affiche_bool
        [|[|false; true; true|];
        [|true; false; false|];
        [|true; true; false|]|];;
|F|V|V|
|V|F|F|
|V|V|F|
- : unit = ()
```

A.2 Proposer une fonction filtre de type ('a -> bool) -> 'a list -> 'a list, de sorte que filtre p l renvoie la liste des éléments de l satisfaisant le prédicat p.

A.3 Définir une fonction deplace : int -> int * int -> (int * int) list telle que deplace n (i, j) renvoie la liste des cases de $\mathcal{E}_n$ auxquelles un cavalier en position (i, j) peut accéder en un coup (*i.e.* la liste des voisines de (i, j)).

Partie B – Traitement fonctionnel récursif

Dans cette partie, une *configuration* est une fonction de type int * int -> bool.

Par souci de simplicité, on déclare globalement la taille de l'échiquier (dans cette partie uniquement), par exemple :

```
# let n_global = 12;;
n_global : int = 12
```

On pourra utiliser dans cette partie la fonction suivante :

```
# let deplace_n_declare case = case :: (deplace n_global case);;
deplace_n_declare : int * int -> (int * int) list = <fun>
```

B.1 Étant donné q configurations $f_1, \ldots, f_q$, on appelle *fusion* de ces configurations la configuration h telle que pour tout (i, j),

$$h(i, j) = f_1(i, j) \vee f_2(i, j) \vee \cdots \vee f_q(i, j),$$

où $\vee$ désigne le ou logique.

a Programmer une fonction fusion_simple permettant de réaliser la fusion de deux configurations.

b Programmer une fonction fusion permettant la fusion des termes d'une liste de configurations.

Un **bonus** sera attribué en cas de programmation récursive terminale (éventuellement appliquée à une fonction auxiliaire).

B.2 Proposer une fonction etend de type (int * int -> bool) -> int * int -> bool qui, à une configuration f, associe la configuration g telle que $g(i, j)$ soit vrai si et seulement si $f(i, j)$ l'est ou (i, j) est voisine d'une case (i', j') telle que $f(i', j')$ soit vraie.

B.3 Proposer une fonction récursive accessibles1 : int * int -> int -> int * int -> bool répondant au problème posé, c'est-à-dire telle que accessibles1 (i0, j0) p renvoie une configuration f telle que pour tout $(i, j) \in [\![0, n-1]\!]^2$, $f(i, j)$ est vraie si et seulement si un cavalier placé en (i_0, j_0) peut atteindre (i, j) en au plus p coups.

Partie C – Traitement impératif et récursif

Dans cette partie et la suivante, on modélise les configurations possibles de l'échiquier par une matrice de booléens de taille $n \times n$. Dans la suite, il est interdit d'employer la fonction deplace_n_declare.

C.1 Proposer une fonction init de type int -> bool vect vect qui à un entier naturel non nul n associe la matrice de taille $n \times n$ dont tous les termes valent false.

C.2 Écrire une fonction valeur : 'a vect vect -> int * int -> 'a qui prend en argument t et un couple (i, j), et renvoie la valeur du terme de t en position (i, j) **(bien revoir la syntaxe donnée en première page avant de répondre).**

C.3 Définir une fonction transmis : int * int -> bool vect vect -> bool, qui, à un couple (i, j) et une configuration e, associe true si (i, j) est voisin d'un couple (i', j') tel que le terme de e en position (i', j') soit vrai, et associe false sinon.

C.4 Écrire une fonction atteintes : bool vect vect -> bool vect vect qui, à une configuration f associe la configuration g donnée par : $g(i, j)$ est vrai si et seulement si $f(i, j)$ l'est ou (i, j) est voisin d'un (i', j') tel que $f(i', j')$ soit vrai.

C.5 Répondre au problème posé avec cette modélisation, de manière récursive, grâce à une fonction accessibles2 de type int -> int * int -> int -> bool vect vect.

Partie D – Traitement purement impératif

On reprend la modélisation précédente, mais on veut une programmation purement impérative. On cherche également un algorithme plus efficace, notamment pour les grandes valeurs de p.

D.1 Définir une fonction echiquier_rempli : bool vect vect -> bool telle que echiquier_rempli config renvoie true si et seulement si tous les termes de config ont pour valeur true.

Remarque : bien sûr, si toutes les cases de l'échiquier sont atteintes en au plus p_0 coups, alors ce sera le cas en au plus p coups, pour tout entier $p \geqslant p_0$. Par exemple, pour l'echiquier classique $\mathcal{E}_8$, peu importe la position de départ, toutes les cases sont atteintes en au plus 6 coups : il serait dommage de ne pas tenir compte de ce résultat si on demande les cases atteintes en au plus 10000 coups ...

D.2 Répondre au problème posé par une fonction accessibles3 de type int -> int * int -> int -> bool vect vect de manière purement impérative, et en tenant compte de la remarque précédente. On pourra utiliser une référence de matrice.

Partie E – Problèmes connexes

E.1 Proposer une fonction pcoups : int -> int * int -> int -> bool vect vect telle que pcoups n (i0, j0) p donne la configuration des cases de l'échiquier de taille $n \times n$ auxquelles on accède en partant de (i_0, j_0) en p coups, mais pas en moins de p coups.

E.2 Proposer une fonction apres_p_coups de type int -> int * int -> int -> bool vect vect, telle que apres_p_coups n (i0, j0) p associe la configuration des positions possibles sur $\mathcal{E}_n$ d'un cavalier parti en (i_0, j_0) après p coups exactement.