

Option Informatique

TP 4: Problème des Reines

David Baelde

david.baelde@ens-lyon.org

1 Réchauffement

Dans le dernier TP je vous proposais d'utiliser certaines fonctions de la librairie standard Caml. Voici un rappel et quelques exercices sur ce sujet. L'utilisation de ces fonctions prédéfinies est à exclure pour les premières questions de concours car cela rendrait la réponse trop facile, évitant la difficulté qu'on vous demande d'aborder. Cependant, sur les fins d'énoncés une fois que vous avez fait vos preuves et plus généralement hors concours, il est bon de connaître certaines fonctions prédéfinies. Cela évite de réinventer la roue, gagnant en temps et en style.

Mémo

```

List.mem e [x1; ...; xn] ≡ e ∈ [x1; ...; xn]
List.map f [x1; ...; xn] ≡ [f x1; ...; f xn]
List.filter f [x1; ...; xn] ≡ [xi | f xi]
List.length [x1; ...; xn] ≡ n
List.rev [x1; ...; xn] ≡ [xn; ...; x1]
List.iter f [x1; ...; xn] ≡ f x1 ; ... ; f xn

```

La dernière fonction est plus compliquée. On donne à la fois une définition explicite et une définition récursive :

```

List.fold_left f i [x1; x2; ...; xn] ≡ f (... (f (f i x1) x2) ...) xn
List.fold_left f i [] ≡ i
List.fold_left f i (tete :: queue) ≡ List.fold_left f (f i tete) queue

```

► Que fait la fonction `(fun l -> List.fold_left (+) 0 (List.map (fun x -> List.fold_left (*) 1 x) l))`? Pensez bien aux espaces autour de `*` pour que Caml n'y lise pas un début de commentaire !

- Soit E un ensemble d'ensembles, représenté par une liste de listes. En utilisant les bonnes fonctions de la librairie standard, calculer

$$\max_{l \in E}(\text{length}(l))$$

- Réécrivez toutes les fonctions du mémo en n'utilisant que `fold_left` !

2 Comprendre le style impératif

Le style fonctionnel est plus conceptuel que l'impératif, il est normal au début de ne pas appréhender facilement toutes ces abstractions. Cependant, le style impératif recèle ses difficultés : il peut être plus difficile de lire du code ainsi écrit, on se retrouve obligé de suivre l'évolution de la mémoire au cours de l'exécution.

- Que calcule la fonction suivante ?

```
let i a b =
  let ab = ref (a,b) in
  let uv = ref (1,0) in
  let qs = ref [] in
  while snd !ab <> 0 do
    let (a,b) = !ab in
    qs := (a/b)::!qs ;
    ab := b, a mod b
  done ;
  while !qs <> [] do
    let q = List.hd !qs in
    let (u,v) = !uv in
    qs := List.tl !qs ;
    uv := (v,u-v*q)
  done ;
  !uv
```

Indice (1) vous l'avez vue en Terminale. Indice (2) la version fonctionnelle est donnée en fin d'énoncé.

3 Le problème des reines

Comme le roi, le taureau aime rentrer dans l'arène.

Le problème des reines est de placer n reines sur un échiquier de taille $n \times n$ sans qu'aucune prise ne soit possible. Les reines sont des reines d'échecs et prennent donc en diagonale, en ligne et en colonne.

On ne connaît pas de technique efficace pour déterminer combien de configurations existent pour un n donné, ni même s'il existe une telle configuration. Il va nous falloir énumérer toutes les possibilités. Le but est de le faire au moins pire, en arrêtant une recherche aussi tôt que possible, et en choisissant la représentation des données qui permettra l'exécution la plus rapide.

On remarque qu'il y a exactement une reine par ligne dans un échiquier solution. Notre stratégie va donc être de tenter de placer une reine sur chaque ligne de 1 à n sans qu'il y ait de menace. Le traitement de la ligne j est de chercher tous les placements encore possibles sur cette ligne en fonction des contraintes sur les lignes précédentes ; pour chaque solution on lance la recherche sur la ligne $j + 1$. Quand on arrive à la ligne $n + 1$ on a gagné.

Pour éviter de recalculer toutes les menaces possibles à chaque fois on va mémoriser les colonnes et les diagonales encore libres à un instant donné. À vous de choisir la représentation efficace de ces données.

- Programmez une fonction `reines` qui prend un entier n et une fonction, et appelle la fonction sur chaque solution du problème en taille n .
- Écrivez une fonction affichant un échiquier, utilisez là pour afficher toutes les solutions en taille 5,6,7,... En utilisant une autre fonction qui dénombre les solutions sans les afficher, jusqu'à quelle taille arrivez-vous ?

Supposons maintenant qu'on ne s'intéresse qu'à l'existence de solutions, on ne veut plus les afficher ni même les compter. Il serait bête de laisser le calcul énumérer toutes les solutions alors qu'il pourrait s'arrêter après le premier succès. Il faudrait que le premier appel de la fonction de succès interrompe l'exécution de `reines`.

Les exceptions vont nous permettre de réaliser cela. Une exception peut être vue comme une erreur. Quand une erreur se produit, le calcul est interrompu. L'erreur traverse toutes les boucles et fonctions en cours, instantanément, pour aller prévenir l'utilisateur. Une seule chose peut l'arrêter : la construction de rattrapage d'exception. La construction `try BODY with RATRAPAGES` permet de lancer le calcul de `BODY` et de rattraper certaines exceptions selon les motifs de `RATRAPAGES`.

Quelques exemples, ne les tapez pas, lisez seulement !

```
# failwith "boum" ;;
Exception: Failure "boum".

# let f x =
  try if x then failwith "bim" else 2 with Failure "bim" -> 42 ;;
val f : bool -> int = <fun>
# f true ;;
- : int = 42
# f (1=2) ;;
- : int = 2

# List.find (fun e -> 1=e) [1;2] ;;
```

```

- : int = 1
# List.find (fun e -> 1=e) [3;2] ;;
Exception: Not_found.
# try Some (List.find (fun e -> 1=e) [2;3]) with Not_found -> None ;;
- : int option = None

```

- En utilisant la bonne fonction passée à `reines`, écrivez une fonction déterminant l'existence de solutions au problème des reines dans une taille donnée.

4 Y'a quelqu'un ?

S'il vous reste un peu de temps, voici encore un problème difficile nécessitant l'énumération d'un grand ensemble de solutions envisageables à filtrer : **Étant donné un ensemble d'entiers S , déterminer l'ensemble de ses parties E telles que :**

$$\forall x \in S, \exists (a, b) \in E^2, a \neq b, x = a - b$$

Cette fois, je ne pense même pas qu'on puisse couper certaines branches de l'énumération comme avant. La recherche sera coûteuse mais la programmation plus simple.

- On représente les ensembles et sous-ensembles par des listes. Écrire une fonction énumérant les parties d'un ensemble, et appelant sur chaque partie une autre fonction passée en argument.
- Étant donnée une liste $[x_1; x_2; \dots; x_n]$ écrire une fonction produisant la liste des $|x_i - x_j|$ pour $i < j$. Programmer ensuite une autre fonction qui, étant donnés deux ensembles A et B , vérifie : $\forall x \in A, |x| \in B$
- Concluez à l'aide des fonctions précédentes.

Voici la version fonctionnelle de la fonction impérative mystère. Elle n'est pas récursive terminale (même si la modification est facile) mais probablement déjà plus efficace pour Caml.

```
let rec f a b =  
  if b = 0 then (1,0) else  
    let (u,v) = f b (a mod b) in  
      (v, u - v*(a/b))
```