TP : Backtracking

17 décembre

1 Tableaux autoréférents

1. Écrire une fonction `somme : int array -> int -> int` telle que l'appel `somme t i` calcule la somme partielle $\sum_{k=0}^{i} t.(k)$ des valeurs du tableau t entre les indices 0 et i inclus.

Un tableau t de $n > 0$ éléments de $[\![0, n-1]\!]$ est dit *autoréférent* si pour tout indice $0 \leq i < n$, $t.(i)$ est exactement le nombre d'occurrences de i dans t, c'est-à-dire que

$$\forall i \in [\![0, n-1]\!], \quad t.(i) = \text{card}\left(\{k \in [\![0, n-1]\!] \mid t.(k) = i\}\right)$$

Ainsi, par exemple, pour $n = 4$, le tableau suivant est autoréférent :

i	0	1	2	3
$t.(i)$	1	2	1	0

En effet, la valeur 0 existe en une occurrence, la valeur 1 en deux occurrences, la valeur 2 en une occurrence et la valeur 3 n'apparaît pas dans t.

2. Justifier rapidement qu'il n'existe aucun tableau autoréférent pour $n \in [\![1; 3]\!]$ et trouver un autre tableau autoréférent pour $n = 4$.
3. Écrire une fonction `est_auto : int array -> bool` qui vérifie si un tableau de taille $n > 0$ est autoréférent. On attend une complexité en $O(n)$.

On propose d'utiliser une méthode de retour sur trace (*backtracking*) pour trouver tous les tableaux autoréferents pour un $n > 0$ donné. Une fonction `gen_auto` qui affiche tous les tableaux autoréférents pour une taille donnée vous est proposée. Cette version ne fonctionne cependant que pour de toutes petites valeurs de n (instantané pour $n = 5$, un peu long pour $n = 8$, sans espoir pour $n = 15$). On pourra vérifier qu'il existe exactement deux tableaux autoréférents pour $n = 4$, un seul pour $n \in \{5, 7, 8\}$ et aucun pour $n = 6$.

Pour accélérer la recherche, il faut élaguer l'arbre (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant pas donner de solution).

4. Que peut-on dire de la somme des éléments d'un tableau autoréférent ? En déduire une stratégie d'élagage pour accélérer la recherche. *Indication : utiliser la fonction* `somme` *de la première question pour interrompre par un échec l'exploration lorsque* `somme t i` *dépasse déjà la valeur maximale possible.*
5. Que peut-on dire si juste après avoir affecté la case $t.(i)$, il y a déjà strictement plus d'occurrences d'une valeur $0 \leq k \leq i$ que la valeur de $t.(k)$? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 15$?
6. Après avoir affecté la case $t.(i)$, combien de cases reste-t-il à remplir ? Combien de ces cases seront complétées par une valeur non nulle ? À quelle condition est-on alors certain que la somme dépassera la valeur maximale possible à la fin ? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 30$?
7. Montrer qu'il existe un tableau autoréférent pour tout $n \geq 7$. *On pourra conjecturer la forme de ce tableau en testant empiriquement pour différentes valeurs de $n \geq 7$. On ne demande pas de montrer que cette solution est unique.*

2 Exceptions

Les exceptions sont une manière d'interrompre une partie d'un programme en cas d'erreur. Par exemple, si vous avez une formule mathématique à calculer, et qu'en plein calcul vous vous apercevez que vous devez diviser 0, vous allez vous arrêter et vous plaindre que la formule n'est pas bien définie. `caml` sait faire pareil.

Les exceptions sont des objets de type `exn` qui ressemblent beaucoup aux types sommes : ce sont des constructeurs, qui peuvent comporter des arguments, et sont déclarés par le mot-clé `exception`.

Quand on a trouvé une erreur, on peut *lancer* une exception avec la fonction `raise` : elle prend une expression en paramètre, et interrompt le calcul (en particulier, tout ce qui devait se passer ensuite dans le programme n'est pas exécuté).

```
#exception Erreur of string;;
let boum () = raise (Erreur "boum");;

try (boum (); "message") with
 | Exit -> "sortie"
 | Erreur message -> "erreur : " ^ message
 | _ -> "exception inconnue"
```

Cela permet de faire des erreurs qui stoppent complètement le calcul. Parfois, on voudrait plutôt détecter l'erreur et utiliser une solution adaptée pour continuer le programme (par exemple si l'erreur est "plus de papier dans l'imprimante", il suffit de demander à l'utilisateur de rajouter du papier avant de continuer, au lieu d'annuler complètement l'impression en cours). On peut *rattrapper* une exception avec la construction `try <expr> with <filtrage>`. Cela se présente un peu comme un `match ... with`, mais le comportement est différent :

— si l'évaluation de `<expr>` ne provoque aucune exception, on renvoie sa valeur
— sinon, on effectue le filtrage sur la valeur de l'exception envoyée

▷ **Question 1.** Écrire une fonction `existe` avec une boucle `for`, tel que `existe elt tableau` teste l'existence de `elt` dans le tableau `tableau` en parcourant le moins possible d'éléments du tableau. ◁

3 N-reines : backtracking

On s'intéresse au problème des N reines. Il s'agit de placer sur un échiquier de taille $N \times N$, N reines sans qu'elles puissent se prendre l'une l'autre. Une solution du problème sera représentée par un tableau de taille N où la case d'indice k contiendra le numéro de la colonne où se trouve la reine de la ligne k s'il en contient une (en effet une ligne contient au plus une reine).

On rappelle que la reine se déplace en ligne, en colonne ou en diagonale.

Le principe de l'algorithme de backtracking est le suivant : on essaie de positionner les reines ligne par ligne en vérifiant qu'une nouvelle reine positionnée ne peut pas prendre une reine préalablement positionnée.

▷ **Question 2.** Ecrire une fonction `check : int-> int array->bool` telle que l'appel à `check k sol` teste si la position de la reine dans la ligne `k` est compatible avec celles des reines en lignes `0` à `k-1`. On suppose que les `k` premières lignes sont convenablement remplies et on vérifie si la `k`eme prolonge la solution partielle ou non. La solution en cours de construction est stockée dans le tableau `sol`. ◁

▷ **Question 3.** Ecrire une fonction `solve : int->int array->bool` telle que `solve n sol` teste si une solution partielle avec une grille remplie jusqu'à la ligne `k-1`, stockée dans `sol`, est prolongeable. La fonction sera récursive et complétera la solution au passage. ◁

▷ **Question 4.** Ecrire une fonction `reines: n->int array option` qui renvoie une solution au problème des reines si elle existe. ◁

4 Sudoku

Les sudokus classiques sont des grilles 9×9 où chaque case est soit blanche, soit contient un chiffre parmi $\{1, \dots, 9\}$. Par convention, les cases blanches contiennent des 0. L'objectif est de remplir les cases blanches, de manière à ce que sur chaque ligne, sur chaque colonne, et sur chacun des 9 carrés 3×3, chaque chiffre apparaît une fois et une seule. Ce sont des sudokus d'ordre 3.

Un sudoku d'ordre n est une grille $n^2 \times n^2$ où chaque case est soit blanche, soit contient un nombre parmi $\{1, \dots, n^2\}$. L'objectif est de remplir les cases blanches, de manière à ce que sur chaque ligne, sur chaque colonne, et sur chacun des n^2 carrés $n \times n$, chaque chiffre apparaît une fois et une seule.

Pour résoudre un sudoku, nous allons utiliser la méthode du backtracking. L'entrée est un entier n et une

grille $n^2 \times n^2$. **Cette grille ne sera pas modifiée**. On procède récursivement, en maintenant une liste des choix effectués, sous la forme de triplet (i, j, k) : ligne i, colonne j, on place l'entier k. L'algorithme procède ainsi : si les conditions du sudoku ne sont pas violées, alors faire un nouveau choix, sinon revenir en arrière, c'est-à-dire modifier le dernier choix qui peut l'être et supprimer les choix faits après celui-ci. S'il n'est pas possible de revenir en arrière, c'est qu'il n'y a pas de solution.

▷ **Question 5.** Coder la fonction `choix_suivant`. Elle déclenche l'exception `Fini` s'il n'y a plus de case libre. ◁

▷ **Question 6.** Coder la fonction `changer_choix`. Elle déclenche l'exception `Echec` s'il n'est pas possible de revenir en arrière. ◁

▷ **Question 7.** Coder la fonction `correct`. Remarquer qu'il est inutile de tester toute la grille, seulement une ligne, une colonne et un carré.
Coder la fonction `retourner_solution`. Tester votre code, par exemple sur une grille vide. ◁

```
let resoudre_sudoku n grille =
let nc = n * n in
let rec boucle liste_choix =
try
 if correct n nc grille liste_choix
  then boucle (choix_suivant nc grille liste_choix)
  else boucle (changer_choix nc grille liste_choix)
with
  | Fini -> retourner_solution nc grille liste_choix
  | Echec -> failwith "pas de solution"
in boucle [] ;;
```