Université Paris Diderot
L3 Informatique

Programmation Fonctionnelle
Année 2016-2017

TP n°8

Deux mises en œuvre d'un resolveur de sudoku : (1) contraintes propositionnelles et sat-solver (2) recherche exhaustive

Le but de ce TP est d'implémenter le *sudoku*. Votre programme devra permettre de resoudre une grille, lue dans un fichier. Deux mises en œuvre sont demandées : dans la première la solution se base sur la production de contraintes propositionnelles et sur l'utilisation d'un sat-solver, fourni, dans la deuxième la solution se base sur la recherche exhaustive.

Règles du jeu

Une grille de sudoku est une matrice carrée 9×9, composée de 9 sous-matrices 3×3. Une instance de sudoku est un remplissage partiel de la grille par des entiers compris entre 1 et 9.

Une solution d'une instance de sudoku est une grille complétement remplie par des entiers compris entre 1 et 9, telle que :

— chaque ligne, colonne et sous-matrice est sans répétition (et contient donc une et une seule fois chaque entier de 1 à 9).

— les cases pré-remplies gardent leurs valeurs.

Voici une instance de sudoku et une de ses solutions [1].

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
	4		1					

2	3	5	4	6	8	7	1	9
4	6	9	1	7	3	2	8	5
7	8	1	9	2	5	3	4	6
1	2	3	5	9	7	4	6	8
6	5	4	8	3	2	1	9	7
8	9	7	6	4	1	5	3	2
5	1	6	2	8	4	9	7	3
9	7	2	3	1	6	8	5	4
3	4	8	7	5	9	6	2	1

Il existe environ $6,67 \times 10^{21}$ solutions distinctes de l'instance vide. Le but de cet exercice est d'écrire un programme qui, pour une instance donnée, trouve une solution, ou signale qu'il n'en existe aucune.

Chargement de la grille

La grille à resoudre est mémorisée dans un fichier qui comporte neuf lignes contenant chacune neuf chiffres décimales, le chiffre 0 désignant la case vide.

Écrire une fonction

1. Par définition, chaque instance de sudoku doit avoir une et une seule solution. Nous ne tenons pas compte de cette contrainte ici.

```
val load : string -> int array array
```

qui prend le nom d'un fichier contenant une grille et renvoie une valeur de type `int array array` correspondante à la grille.

Quelques fonctions utiles :

```
val  open_in           :    string -> in_channel
val  input_line        :    in_channel -> string
val  String.get        :    string -> int -> char
val  Char.code         :    char -> int
val  Array.make_matrix :    int -> int -> 'a -> 'a array array
val  close_in          :    in_channel -> unit
```

Solution 1 : contraintes propositionnelles et sat-solver

Pour utiliser le sat-solver `SAT-MICRO` [2] comme resolveur de sudoku, il faut produire, à partir d'une instance I de sudoku, une formule propositionnelle F_I en forme normale conjonctive telle que toute affectation qui satisfait F_I corresponde à une solution de I. Avant d'aborder le problème de la construction de F_I, il faut produire le sat-solver : le fichier `sat_solver.ml` fournit pour cela un foncteur, `Make`, qui prend en argument un module `Variables_Sudoku` de signature

```
module type VARIABLES = sig
  type t
  val compare : t -> t -> int
end
```

et produit un module

```
module Solver= Make(Variables_Sudoku)
```

fournissant en particulier la fonction :

```
val solve : literal list list -> literal list option
```

où `literal` est le type

```
bool*Variables_Sudoku.t
```

L'argument de la fonction `solve` sera la formule F_I, de type `literal list list`.

Il faut commencer par définir le module `Variables_Sudoku` : type `t` (le type des variables propositionnelles) et fonction `compare`. Le choix approprié pour le type `t` est de decréter qu'une variable est un triplet (i, j, k) d'entiers (compris entre 1 et 9), désignant la proposition "la case de coordonnées (i, j) est remplie par k".

En ce qui concerne la définition de `compare`, il est possible d'utiliser la fonction prédéfinie

```
val compare: 'a -> 'a -> int
```

Une fois le module `Solver` produit [3], il faut produire F_I.

À titre d'exemple, voici une formule qui exprime le fait que la première case de la grille est remplie par *au moins* une valeur, et qu'elle ne'est pas remplie à la fois par la valeur 1 et par la valeur 2 :

```
[
[(true,(1,1,1));(true,(1,1,2));(true,(1,1,3));(true,(1,1,4));(true,(1,1,5));
```

2. SAT-MICRO est du à S. Conchon, J. Kanig et S. Lescuyer. Voir `https://hal.inria.fr/inria-00202831`. Pour plus de détails sur l'algorithme implémenté, *cf.* `https://fr.wikipedia.org/wiki/Algorithme_DPLL`.

3. Pour utiliser le top-level d'OCaml en phase d'écriture du programme, utilisez la directive `#use sat_sover.ml;;`

Pour la phase se test passez au mode compilé pour améliorer les performances.

```
(true,(1,1,6));(true,(1,1,7));(true,(1,1,8));(true,(1,1,9))];
[(false(1,1,1));(false,(1,1,2))]
]
```

Production des contraintes

La fonction principale de cette section est

```
val all_constraints : int array array -> literal  list list
```

qui prend une instance de sudoku I et renvoie F_I. La majorité des contraintes sont en réalité indépendantes de l'instance, et peuvent être produites par les fonctions suivantes, qui modifient une réference declarée par `all_constraints` :

```
val exists_unique : (bool * (int * int * int)) list list ref -> unit
val lines_and_columns : (bool * (int * int * int)) list list ref -> unit
val subsquares : (bool * (int * int * int)) list list ref -> unit
```

Pour les contraintes spécifique de l'instance :

```
val instance_constraints :
int array array -> (bool * (int * int * int)) list list ref -> unit
```

Solution des contraintes et affichage du résultat

Une fois produite la formule F_I, on peut lancer le sat-solver, interpréter la solution renvoyée et l'afficher. Si `f` a type `literal list list`, et vaut F_I, l'appel

```
sat_solver.solve f
```

renvoie une valeur de type `literal list option`. Cette valeur est `None` si F_I n'est pas satisfaisable, `Some l` sinon. Dans ce deuxième cas, `l` est une affectation qui représente une solution de I : si `(true,(i,j,k))` est dans `l`, alors il faut remplir la case de coordonnées `(i,j)` par `k`.

Écrire une fonction

```
val instance_of_affectation : literal list -> int array array
```

qui prend l'affectation renvoyée par le sat-solver et renvoie la solution correspondante.

Écrire une fonction

```
val print : int array array -> unit
```

pour afficher une grille.

Écrire une fonction

```
val one_solution : string -> unit
```

qui prend le nom d'un fichier et affiche une solution de l'instance de sudoku memorisée dans ce fichier.

Solution 2 : recherche exhaustive, backtracking et récurrence

Backtracking

La recherche exhaustive d'une solution pour une instance de sudoku consiste en la construction d'un arbre modélisant les extensions possibles de la grille donnée. On fixe tout d'abord une énumération des cases qui donne l'ordre dans lequel on cherchera à remplir la grille initiale. L'arbre d'exploration est alors défini comme suit :

— La racine est étiquetée par l'instance de sudoku initialement donnée,

— Les fils d'un nœud étiqueté G sont étiquetés par toutes les grilles $G_1, ..., G_k$ obtenues en remplissant correctement – c'est-à-dire en respectant les contraintes du sudoku – la prochaine case vide de la grille G, selon l'énumération fixée initialement. Notons qu'il y en a au plus 9.

Pour les feuilles de cet arbre, deux cas sont possibles :

— Il n'y a plus de cases à remplir : la feuille correspondante est une solution de l'instance initiale du sudoku.

— Aucun remplissage de la case à remplir ne satisfait les règles : la grille partiellement remplie déjà produite ne peut mener à une solution.

Mise en œuvre

Le coeur du programme est une fonction récursive,

```
val exhaustive_search : int array array  -> int * int -> int -> bool
```

qui prend en argument

— la grille partiellement remplie,

— les coordonnés de la prochaine case à remplir,

— la valeur à écrire dans cette case

qui vérifie si la case donnée peut être remplie par la valeur donnée, l'écrit si c'est le cas, et passe la main à la prochaine case, ou éventuellement à la même case et à la prochaine valeur. Cette fonction retournera un booléen, les cas de bases étant le renvoie de

— `false` si le troisième parametre est 10 (pas de valeurs possibles pour la case).

— `true` se le deuxième paramètre est la "première case à l'exterieur de la grille" (solution trouvée).

1. Écrire une fonction

   ```
   val possible : int array array -> int * int -> int -> bool
   ```

   qui prend en paramètre une grille, les coordonnées d'une case et un entier entre 1 et 9 et qui teste si l'ajout de cet entier aux coordonnées données mène à une grille satisfaisant les contraintes du sudoku.

2. Écrire une fonction

   ```
   val solve : string -> unit
   ```

   qui prend le nom d'un fichier et affiche une solution de l'instance de sudoku memorisée dans ce fichier, mettant en œuvre la recherce exhaustive décrite ci-dessus. La fonction `solve` peut déclarer localement la fonction auxiliaire `exhaustive_search` décrite ci-dessus.