Haskell (IR3) – Sat Solver

Stéphane Vialette

21 novembre 2018

Le but de ce TP est de développer en Haskell un solveur simple pour les formules booléennes (*Sat Solver*). C'est un problème très important, tout autant pour l'informatique théorique que pour les très nombreuses applications.

En informatique, le problème SAT est un problème de décision défini par des formules logiques. Il s'agit, étant donnée une formule de logique propositionnelle, de décider si cette formule possède une solution, c'est-à-dire s'il existe une assignation des variables rendant la formule vraie.

Une *clause* est une proposition de la forme $\bigvee_{i=1}^{n} \ell_i = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$ où les ℓ_i sont des *littéraux* (*positifs* ou *négatifs*). Une formule du calcul propositionnel est en *forme normale conjonctive* (ou *forme clausale*) si elle est une conjonction de clauses.

Exemple. Soit l'ensemble de variables $\{x_1, x_2, x_3\}$ et la formule $f = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_1)$. La formule f est satisfaisable puisque, si on pose $x_1 = \textit{vrai}$, $x_2 = \textit{faux}$, $x_3 = \textit{vrai}$, alors f est logiquement vrai (c'est facile à vérifier !). En revanche, la formule $f' = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$ n'est pas satisfaisable. En effet, f' sera évalué à faux quelles que soient les valeurs attribuées à x_1, x_2 et x_3 (c'est par contre un peu plus difficile à vérifier ! . . . mais pas trop !).

Les modules **`Var`** (complet), **`Lit`** (complet), **`Clause`** (incomplet), **`Fml`** (incomplet) sont fournis et représentent respectivement les variables, les littéraux, les clauses et les formules.

Question 1: Des clauses
Compléter le module **`Clause`**.

Question 2: Des formules
Compléter le module **`Fml`**.

Question 3: Sat Solver
Il s'agit maintenant d'écrire un sat-solveur (module **`SatSolver`**).

Quelques mots sur la stratégie que nous avons suivre. Nous allons récursivement décider les littéraux vrais et faux. À chaque étape nous choisirons un littéral dans une clause unitaire (si une telle clause existe) et sinon un littéral avec le plus grand nombre d'occurrences.

(a) Écrire la fonction

```
selectLiteral :: F.Fml a -> Maybe Lit.Lit a
```

qui retourne (i) un littéral apparaissant dans une clause unitaire (si une telle clause existe dans la formule) ou (ii) un littéral qui apparaît le plus grand nombre de fois. Si la formule est vide, la fonction retourne **`Nothing`**.

(b) Revenir au module `Clause` et écrire la fonction

```
reduce :: Lit.Lit a -> Clause.Clause a -> Clause.Clause a
```

qui réduit une clause relativement à un littéral (la fonction retourne une nouvelle clause). Réduire une clause relativement à un littéral ℓ est l'opération consistant à supprimer toutes les occurrences du littéral opposé $\neg\ell$.

(c) Revenir au module `Fml` et écrire la fonction

```
reduce :: Lit.Lit a -> Fml a -> Fml a
```

qui réduit une formule relativement à un littéral. (la fonction retourne une nouvelle formule). Réduire une formule relativement à un littéral ℓ est l'opération consistant à (i) réduire toutes les clauses de la formule relativement à ℓ et (ii) supprimer toutes les clauses qui contiennent le littéral ℓ.

(d) Revenir au module `Fml` et écrire la fonction

```
solve :: Fml.Fml a -> Maybe (Map.Map (Var.Var a) Bool)
```

qui retourne une assignation des variables rendant la formule vraie (sous forme d'une map `Map.Map (Var.Var a) Bool`) ou `Nothing` si la formule n'est pas satisfaisable.

Écrire la fonction

```
solve' :: Fml.Fml a -> Maybe [Lit.Lit a]
```

qui retourne un ensemble de littéraux rendant la formule vraie ou `Nothing` si la formule n'est pas satisfaisable.

(e) Écrire la fonction

```
solveAll :: Fml.Fml a -> Maybe [Map.Map (Var.Var a) Bool]
```

qui retourne toutes les assignations des variables rendant la formule vraie ou `Nothing` si la formule n'est pas satisfaisable.

Écrire la fonction

```
solveAll' :: Fml.Fml a -> Maybe [[Lit.Lit a]]
```

qui retourne tous les ensembles de littéraux rendant la formule vraie ou `Nothing` si la formule n'est pas satisfaisable.

(f) Écrire la fonction

```
maxSolve :: Fml.Fml a -> Map.Map (Var.Var a) Bool
```

qui retourne une assignation des variables rendant vraie le plus grand nombre possible de clauses de la formules.

Écrire la fonction

```
maxSolve' :: Fml.Fml a -> [[Lit.Lit a]]
```

qui retourne tous les ensembles de littéraux rendant vraie le plus grand nombre possible de clauses de la formules.