Haskell (Info 3) – Sat Solver

Fabian Reiter (`fabian.reiter@univ-eiffel.fr`)
Stéphane Vialette (`stephane.vialette@univ-eiffel.fr`)

31 octobre 2020

Le but de ce TP est de développer en Haskell un solveur simple pour les formules booléennes (on appelle un tel outil un *Sat Solver*). Résoudre des formules booléennes est un problème fondamental en informatique, de l'informatique théorique aux (nombreuses) applications.

En informatique, le problème SAT est un problème de décision défini par des formules logiques. Étant donnée une formule de logique propositionnelle sous forme normale conjonctive, il s'agit de décider si cette formule possède une solution, c'est-à-dire s'il existe une assignation des variables qui rend vraie la formule.

Un peu de vocabulaire. 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* (CNF) si elle est une conjonction de clauses. Clairement, pour satisfaire une formule CNF il suffit de satisfaire au moins un littéral de chaque clause. Un exemple devrait clarifier tout ça.

> Considérons l'ensemble de variables $X = \{x_1, x_2, x_3\}$ et la formule $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_1)$. La formule ϕ 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 $\phi' = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$ n'est pas satisfaisable. En effet, ϕ' sera évalué à faux quelles que soient les valeurs attribuées aux variables x_1, x_2 et x_3 (c'est par contre un peu plus difficile à vérifier ! ... mais pas trop ! ... et de toute façon nous aurons bientôt un outil pour tester ça).

Les modules

- **`Data.Algorithm.SatSolver.Var`** (représentation et manipulation des variables, il est donné complet),
- **`Data.Algorithm.SatSolver.Lit`** (représentation et manipulation des littéraux, il est donné complet),
- **`Data.Algorithm.SatSolver.Clause`** (représentation et manipulation des clauses, il est donné incomplet),
- **`Data.Algorithm.SatSolver.Fml`** (représentation et manipulation des formule, il est donné incomplet),

sont fournis. Une fois ces modules terminés, vous aurez à écrire complètement le module **`Data.Algorithm.SatSolver.Solver`** (ce sera finalement assez simple à l'aide des fonctions définies dans les modules **`Var`**, **`Lit`**, **`Clause`** et **`Fml`**). Notez la présences des modules

- **`Data.Algorithm.SatSolver.Var.Some`** (quelques variables),
- **`Data.Algorithm.SatSolver.Lit.Some`** (quelques littéraux),

— `Data.Algorithm.SatSolver.Clause.Some` (quelques clauses),
— `Data.Algorithm.SatSolver.Fml.Some` (quelques formules),
qui vous permétront d'expérimenter et de tester vos fonctions.

Un (très court) module `Data.Algorithm.SatSolver.Utils` est également fourni.

La documentation ainsi que la correction vous sont données. Nous allons utiliser des modules (obligatoire !) et `stack` pour gérer notre projet (obligatoire également !). Le projet s'appelera `satsolver` et les différents modules résident dans `Data.Algorithm.SatSolver`.

Question 1: Création du projet

Création et gestion du projet avec `stack` (`https://docs.haskellstack.org/en/stable/README/`).

Pour résumer (mais juste les grandes lignes),

```
$ stack new satsolver
...
$ cd satsolver
$ mkdir src/Data
$ mkdir src/Data/Algorithm
$ mkdir src/Data/Algorithm/SatSolver
```

Un peu de ménage,

```
$ rm src/Lib.hs
$ rm app/Main.hs
```

Editez `package.yaml`.

— Commentez (à l'aide du caractère `#`) l'intégralité de la section `executables`. (Vous pourrez toujours expérimenter la compilation d'un exécutable plus tard.)
— Remplissez l'entête du fichier (nom, ...).
— Ajoutez une sous-section `exposed-modules` à la section `library`. C'est là que vous mettrez vos différents modules.

À la fin du TP, la section `library` devrait ressembler à çà :

```
library:
  source-dirs: src
  exposed-modules:
    - Data.Algorithm.SatSolver.Clause
    - Data.Algorithm.SatSolver.Clause.Some
    - Data.Algorithm.SatSolver.Fml
    - Data.Algorithm.SatSolver.Fml.Some
    - Data.Algorithm.SatSolver.Lit
    - Data.Algorithm.SatSolver.Lit.Some
    - Data.Algorithm.SatSolver.Solver
    - Data.Algorithm.SatSolver.Utils
    - Data.Algorithm.SatSolver.Var
    - Data.Algorithm.SatSolver.Var.Some
```

Pour compiler votre projet :

```
$ stack build
```

Pour générer la documentation :

```
$ stack haddock
```

Pour lancer l'interpréteur et rendre vos modules accessibles :

```
$ stack exec -- ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/vialette/.ghci
>>> import qualified Data.Algorithm.SatSolver.Var as Var -- par exemple
>>> Var.mk "x1"
"x1"
```

Pour lancer les tests (optionnel, il faut implémenter dans `test`) :

```
$ stack test
```

Question 2: Des variables

Lire et comprendre le (très court !) module **Data.Algorithm.SatSolver.Var**. Le module **Data.Algorithm.SatSolver.Var.Some** fournit quelques exemples d'utilisation.

Question 3: Des littéraux

Lire et comprendre le (court !) module **Data.Algorithm.SatSolver.Lit**. Le module **Data.Algorithm.SatSolver.Lit.Some** fournit quelques exemples d'utilisation.

Question 4: Des clauses

Compléter le module **Clause**. Le module **Data.Algorithm.SatSolver.Clause.Some** fournit quelques exemples d'utilisation.

Question 5: Des formules

Compléter le module **Fml**. Le module **Data.Algorithm.SatSolver.Fml.Some** fournit quelques exemples d'utilisation.

Les fonctions `toNormal :: (Ord a, Ord b ,Num b, Enum b) => Fml a -> Fml b` et `toDIMACSString` ne sont pas à écrire dans un premier temps.

Question 6: Sat Solver

Il s'agit maintenant d'écrire un module (**Data.Algoritm.SatSolver.Solver**) pour résoudre les formules.

Quelques mots sur l'algorithme récursif assez simple que nous avons utiliser. Choisir un littéral ℓ qui apparaît dans la formule (voir ci-après pour les détails) et essayer de résoudre récursivement la formule en choisissant ce littéral. Si nous échouons, essayer de résoudre récursivement la formule en choisissant $\neg\ell$[1]. Si nous échouons à nouveau (c'est-à-dire si obtenons une clause vide), la formule n'est pas satisfaisable.

Les règles pour choisir un littéral sont les suivantes :

1. S'il existe une clause unitaire, choisir le littéral qui apparaît dans cette clause unitaire.
2. Sinon, s'il existe un littéral qui apparaît seulement négativement ou positivement, choisir ce littéral.
3. Sinon, choisir le littéral qui apparaît avec le plus grand nombre d'occurrences.

(a) Écrire la fonction

```
reduceClause :: (Eq a, Ord a) =>
  Lit.Lit a -> Clause.Clause a -> Clause.Clause a
```

1. Si une formule est satisfaisable et si x est une variable de ϕ alors la formule est satisfaire pour le littéral x ou pour le littéral $\neg$x . . . c'est immédiat !

qui, pour un littéral ℓ et une clause c, retoune la clause c′ obtenue à partir de c en supprimant le ou les occurrences de $\neg\ell$.

(b) Écrire la fonction

```
reduceFml :: (Eq a, Ord a) => Lit.Lit a -> Fml.Fml a -> Fml.Fml a
```

qui, pour un littéral ℓ et une formule CNF ϕ, retoune la formules ϕ' obtenue à partir de ϕ en supprimant le ou les occurrences de $\neg\ell$ dans toutes les clauses de ϕ.

(c) Écrire la fonction

```
selectLit :: (Ord a) => Fml.Fml a -> Maybe (Lit.Lit a)
```

qui, pour une formule CNF ϕ, retourne un littéral en utilisant les règles données plus haut. Si la formule ne contient aucun littéral, la fonction retourne **Nothing**.

(d) Écrire pour terminer la fonction

```
solve :: (Ord a) => Fml.Fml a -> Maybe (Assignment a)
```

qui, pour une formule CNF ϕ, retourne une affectation qui rend vraie ϕ (si ϕ est satisfaisable), et **Nothing** sinon.

Une affectation est représentée par le type **Assignment** a qui est est défini par

```
type Assignment a = M.Map (Var.Var a) Bool
```

Et voilà !

Question 7: Pour aller plus loin.

Le format DIMACS est un standard pour réprésenter les formules CNF. C'est un format très simple (consulter par exemple `https://fairmut3x.wordpress.com/2011/07/29/cnf-conjunctive-normal-form-dimacs-format-explained/`).

— Les lignes qui débutent par `c` sont des commentaires.

— Il y a exactement une ligne qui commence par `p`. Cette ligne résume la formule et a le format suivant

```
p cnf number-of-variables number-of-clauses
```

— Chaque clause est représentée par des entiers (positifs ou négatifs) séparés par des espace. La ligne se termine toujours par `0`.

Par exemple, nous pouvons représenter la formule

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee x_6) \wedge (\neg x_6 \vee \neg x_7) \wedge (x_7 \vee x_8) \wedge (\neg x_8 \vee \neg x_9) \wedge (x_9 \vee \neg x_1)$$

de la façon suivante :

```
c haskell rule
p cnf 9 9
+1 +2 0
-2 -3 0
+3 +4 0
-4 -5 0
+5 +6 0
-6 -7 0
+7 +8 0
-8 -9 0
-1 +9 0
```

Cela fait sens si les littéraux sont des entiers signés mais nos types sont les suivants (en vrac) :

```
newtype Var a = Var { getName :: a } deriving (Eq, Ord)
data Lit a = Neg (Var.Var a) | Pos (Var.Var a) deriving (Eq, Ord)
newtype Clause a = Clause { getLits :: [Lit.Lit a] } deriving (Eq, Ord)
newtype Fml a = Fml { getClauses :: [Clause.Clause a] }
```

Il conviendra donc de paramétrer nos types pour des numériques.

(a) Écrire la fonction

```
toNormal :: (Ord a, Ord b ,Num b, Enum b) => Fml a -> Fml b
```

du module `Data.Algorithm.SatSolver.Fml` qui transforme toute formule CNF en une formule équivalente où les variables sont définies sur des numériques.

(b) Écrire maintenant la fonction

```
toDIMACSString :: (Ord a) => Fml a -> String
```

qui retourne la représentation au format DIMACS d'une formule.