Haskell (IR3) – Sat Solver

Stéphane Vialette

18 janvier 2015

Le but de ce TP est de développer en Haskell un solveur simple pour les formules booléennes (Sat Solver). C'ets un problème très important tout autant poru l'informatique théorique que pour les 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 $\bigvee_{i=1}^{n} v_i = v_1 \vee v_2 \vee \ldots \vee v_n$ où les v_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 $\{v_1, v_2, v_3\}$ et la formule $f = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee \neg v_1)$. f est satisfaisable puisque, si on pose $v_1 = \textit{vrai}$, $v_2 = \textit{faux}$, $v_3 = \textit{vrai}$, alors f est logiquement vrai (c'est facile à vérifier !. En revanche, $f' = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee v_1) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_3)$ n'est pas satisfaisable, car f' sera évalué comme faux quelles que soient les valeurs attribuées à v_1, v_2 et v_3 (c'est par contre un peu plus difficile à vérifier !).

Nous définissons les types suivants (définis dans les modules **Variable**, **Literal**, **Clause**, **Formula**) :

Variable.hs

```
module Variable
(
  Variable
)
where

-- | 'Variable' type
type Variable = Int
```

Literal.hs

```
module Literal
(
  Literal
, isPositiveLiteral
, isNegativeLiteral
, toVariable
```

```
)
where

import qualified Variable as V

-- | 'Literal' type
type Literal = Int

...
```

Clause.hs

```
module Clause
(
  Clause
, makeClause
, isEmpty
, size
, isUnit
)
where

import qualified Data.List as List
import qualified Literal as L

-- | 'Clause' type
type Clause = [L.Literal]

...
```

Formula.hs

```
module Formula
(
  Formula(..)
, makeFormula
, isEmpty
, isSatisfied
, unitClauses
, mostFrequentLiteral
, hasUnsatisfiedClause
, makeExampleFormula
)
where

import qualified Data.Ord as Ord
import qualified Data.List as List
```

```haskell
import qualified Literal as L
import qualified Variable as V
import qualified Clause as C

-- | 'Formula' type
type Formula = [C.Clause]

...
```

Question 1: Des litéraux

(a) Écrire les fonctions

`isPositiveLiteral :: Literal -> Bool`

et

`isNegativeLiteral :: Literal -> Bool`

qui retournent `True` si le litéral passé en argument est positif ou négatif, resptivement.

(b) Écrire la fonction

`toVariable :: Literal -> V.Variable`

qui retourne la variable associée à un litéral.

Question 2: Des clauses

(a) Écrire la fonction

`makeClause :: [L.Literal] -> Clause`

qui construit une clause à partir d'une liste de litéraux. Les occurrences multiples d'un même litéral doivent être réduites à une unique occurrence et 0 n'est pas un litéral valide (pour simplifier il sera simplement ignoré)

```
*Clause: makeClause []
[]
*Clause: makeClause [1..3]
[1,2,3]
*Clause: :t makeClause [1,2,3]
makeClause [1,2,3] :: Clause
*Clause: makeClause [1,-2,3]
[-2,1,3]
*Clause: makeClause [1,-2,3,3,-2]
[-2,1,3]
*Clause: makeClause [1,-2,3,3,-2,0]
[-2,1,3]
*Clause:
```

(b) Écrire la fonction

`isEmpty :: Clause -> Bool`

qui retourne `True` si la clause ne contient aucun argument.

(c) Écrire la fonction

`size :: Clause -> Int`

qui retourne le nombre de litéraux dans la clause.

(d) Écrire la fonction

`isUnit :: Clause -> Bool`

qui retourne `True` si la clause est unitaire (*i.e.*, ne contient qu'un seul litéral).

Question 3: Des formules

(a) Écrire la fonction

```
makeFormula :: [C.Clause] -> Formula
```

qui construit une clause à partir d'une liste de clauses. Les occurrences multiples d'une même clause doivent être réduites à une unique occurrence.

(b) Écrire la fonction

```
isEmpty :: Formula -> Bool
```

qui retourne `True` si la clause ne contient aucun litéral.

(c) Écrire la fonction

```
isSatisfied :: Formula -> Bool
```

qui retourne `True` si la clause est satisfaite (*i.e.*, ne contient aucune clause). Attention cela est très différent du cas où la formule contient une clause vide, puisque dans ce cas la formule ne peut pas être satisfaite.

```
*Formula: isSatisfied $ makeFormula [[1,2],[3,4]]
False
*Formula: isSatisfied $ makeFormula [[1,2],[]]
False
*Formula: isSatisfied $ makeFormula [[]]
False
*Formula: isSatisfied $ makeFormula []
True
*Formula:
```

(d) Écrire la fonction

```
unitClauses :: Formula -> Formula
```

qui retourne une nouvelle formule constituée des clauses unitaires uniquement.

```
*Formula: unitClauses $ makeFormula []
[]
*Formula: unitClauses $ makeFormula [[1,2],[3],[4,5,6],[7]]
[[3],[7]]
*Formula: unitClauses $ makeFormula [[1],[2],[3]]
[[1],[2],[3]]
*Formula:
```

(e) Écrire la fonction

```
mostFrequentLiteral :: Formula -> L.Literal
```

qui retourne un litéral ayant le plus grand nombre d'occurrences. La formule doit être non vide.

```
*Formula: mostFrequentLiteral $ makeFormula [[1,2],[-2,3],[2,3,4]]
3
*Formula:
```

(f) Écrire la fonction

```
hasUnsatisfiedClause :: Formula -> Bool
```

qui retourne `True` si la formule contient une clause non satisfaite.

```
*Formula: hasUnsatisfiedClause $ makeFormula []
False
*Formula: hasUnsatisfiedClause $ makeFormula [[]]
True
*Formula: hasUnsatisfiedClause $ makeFormula [[1,2],[]]
True
*Formula:
```

(g) Écrire la fonction

```
size :: Formula -> Int
```

qui retourne le nombre de clauses dans la formule.

Question 4: Sat Solver

(a) Écrire le module `SatSolver`. Notez que les fonctions `selectLiteral`, `reduceFormula` et `reduceClause` ne sont exportées que pour des raisons pédagogiques, elles ne devraient pas l'être.

```
module SatSolver
(
  selectLiteral
, reduceFormula
, reduceClause
, solve
)
where

import qualified Data.List as List
import qualified Data.Map as Map
import qualified Data.Ord as Ord

import qualified Variable as V
import qualified Literal as L
import qualified Clause as C
import qualified Formula as F

...
```

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

(b) Écrire la fonction

```
selectLiteral :: F.Formula -> L.Literal
```

qui retourne (i) un litéral apparaissant dans une clause unitaire (si une telle clause existe dans la formule) ou (ii) un litéral qui apparaît le plus grand nombre de fois. La formule doit être non vide.

```
*SatSolver: selectLiteral $ F.makeFormula [[1,2],[3],[1,3]]
3
*SatSolver: selectLiteral $ F.makeFormula [[1,2],[2,-3],[1,3]]
2
*SatSolver:
```

(c) Écrire la fonction

```
reduceClause :: L.Literal -> C.Clause -> C.Clause
```

qui réduit une clause relativement à un litéral. La fonction retourne une nouvelle clause.

```
*SatSolver: reduceClause 1 $ C.makeClause [1,-2,3,-4]
[-4,-2,1,3]
*SatSolver: reduceClause (-1) $ C.makeClause [1,-2,3,-4]
[-4,-2,3]
*SatSolver: reduceClause (-1) $ C.makeClause [1,-2,3,-4,1]
[-4,-2,3]
*SatSolver:
```

(d) Nous sommes maintenant armés poru écrire le fonction Écrire la fonction

```
solve :: F.Formula -> Maybe (Map.Map V.Variable Bool)
```

qui retourne une assignation des variables rendant la formule vraie (sous forme d'un `Just Map.Map V.Variable Bool`) ou `Nothing` sinon. Pour faciliter les tests, écrire une fonction

```
makeExampleFormula :: Formula
```

dans le module `Formula`.

```
*SatSolver: F.makeExampleFormula
[[-3,1],[-5,1,4],[2,3,4],[-5,-3,-1],[4,5],[-4,-3,5],[2],[-2,1]]
*SatSolver: solve F.makeExampleFormula
Just (fromList [(1,True),(2,True),(3,False),(5,True)])
*SatSolver: solve $ F.makeFormula [[1,2],[1,-2],[-1,2],[-1,-2]]
Nothing
*SatSolver: solve $ F.makeFormula []
Just (fromList [])
*SatSolver: solve $ F.makeFormula [[]]
Nothing
*SatSolver:
```