Haskell (IR3) – Sat Solver

Stéphane Vialette

25 janvier 2015

Lire le sujet en entier avant de commencer.

1 Rappels

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 !).

2 Introduction

Le but de ce TP noté est de développer en Haskell un solveur simple pour les formules booléennes (Sat Solver). Il se place donc dans la continuité du TP 4. L'objectif est ici double :

1. Utiliser de nouvelle définitions pour les types `Variable`, `Literal`, `Clause` et `Formula`.
2. Améliorer l'algorithme de résolution.

Commençons par quelques remarques sur les types `Variable`, `Literal`, `Clause` et `Formula` tels qu'ils ont été définis dans le TP 4 :

```
type Variable = Int
type Literal  = Int
type Clause   = [Literal]
type Formula  = [Clause]
```

Il a été remarqué en TP que 0 n'est pas une valeur valide pour une variable (car il est impossible de différencier -0 de $+0$ et donc de différencier les littéraux littéraux). De plus ces définitions ne facilitent pas leur utilisation dans les cas pratiques : à quels entiers dois je associer les variables x_{riri}, x_{fifi}, x_{loulou} ? Il est ici nécessaire d'introduire un objet faisant correspondre les chaînes « riri », « fifi" » et « loulou" » aux entiers $\{1, 2, 3\}$, par exemple, ce qui complique considérablement l'utilisation de notre bibliothèque.

Afin de faciliter l'utilisation de la bibliothèque, nous allons utiliser les types algébriques suivant :

```haskell
-- Boolean variable data type
data Variable a = Variable a
                  deriving (Show, Eq, Ord)

-- Boolean literal data type
data Literal a = Negative (V.Variable a) | Positive (V.Variable a)
                 deriving (Show, Eq, Ord)

-- Clause (disjunction of literals) data type
data Clause a = Clause [L.Literal a]
                deriving (Show, Eq, Ord)

-- CNF formula data type
data Formula a = Formula [C.Clause a]
                 deriving (Show)
```

Vous remarquerez qu'à l'aide de ces types, il est désormais possible d'écrire les expressions suivantes (par exemple en se focalisant sur les variables) :

```
>>> import qualified Variable as V
>>>
>>> V.Variable 0
Variable 0
>>> V.Variable "riri"
Variable "riri"
>>> V.Variable [1..10]
Variable [1,2,3,4,5,6,7,8,9,10]
```

N'oubliez pas que les constructeur de valeurs (par exemple `Positive` et `Negative` dans la définition du type `data Literal = Positive a | Negative a ...`) s'utilisent à la fois pour le filtrage (pattern matching) et comme des fonctions. Un exemple d'utilisation en annexe (2.1).

Question 1: Comprendre

Il ne s'agit pas à proprement parler ici d'une question. Parcourir les modules `Variables`, `Literal`, `Clause`, `Formula`, et `SatSolver` pour comprendre la nouvelle implémentation. La documentation fournie des différents modules peut vous aider à lever les ambiguïtés.

Reportez vous une nouvelle fois à l'Annexe 2.1 pour vous familiariser à nouveau avec la manipulation des nouveaux types.

Question 2: Résolution : L'algorithme de Davis–Putnam–Logemann–Loveland (DPLL)

Nous allons modifier nos modules pour implémenter l'algorithme de résolution de Davis–Putnam–Logemann–Loveland. Cet algorithme concerne essentiellement (dans notre implémentation) le choix du litéral à résoudre. Elle se base sur deux notions : la propagation unitaire et l'élimination des littéraux purs.

La propagation unitaire Pour chaque clause unitaire (ne contenant qu'un littéral), on supprime les clauses comprenant ce littéral, et on enlève le littéral opposé des autres clauses. Par exemple, une clause (x) montre que x est vrai, on peut supprimer toutes les clauses de la forme $A \vee x$ et supprimer tous les $\overline{x}$ des autres clauses. Cette règle peut se répéter jusqu'à ce qu'il ne reste plus de clause unitaire.

L'élimination des littéraux purs Si une variable apparaît exclusivement sous la forme de littéral positif (ou de littéral négatif) dans l'ensemble de clauses, on peut supprimer toutes

les clauses dans lesquelles elle apparaît. Par exemple, si on ne trouve aucune clause contenant $\overline{x}$, toutes les clauses contenant x peuvent être supprimées. Là encore, cette règle peut se répéter et se combiner.

Nous réécrivons donc la fonction `selectLiteral` du module **`SatSolver`** de la façon suivante :

```
selectLiteral :: (Eq a, Ord a) => F.Formula a -> L.Literal a
selectLiteral f
  | (not . List.null) cs = (List.head . C.literals . List.head) cs
  | (not . List.null) ls = (List.head) ls
  | otherwise            = F.mostFrequentLiteral f
  where
    cs = F.unitClauses f
    ls = F.pureLiterals f
```

(a) Écrire la fonction

```
positiveLiterals :: (Eq a) => Formula a -> [L.Literal a]
```

du module **`Formula`** qui retourne la liste des littéraux positifs qui apparaissent dans une formule. Un littéral doit apparaître au plus une fois dans une telle liste.

(b) Écrire la fonction

```
negativeLiterals :: (Eq a) => Formula a -> [L.Literal a]
```

du module **`Formula`** qui retourne la liste des littéraux négatifs qui apparaissent dans une formule. Un littéral doit apparaître au plus une fois dans une telle liste.

(c) Écrire la fonction

```
pureLiterals :: (Eq a) => Formula a -> [L.Literal a]
```

du module **`Formula`** qui retourne la liste des littéraux pures qui apparaissent dans une formule. Tester votre code (en particulier en utilisant la fonction `selectLiteral` du module **`SatSolver`**).

Question 3: Résolution : Vers l'algorithme de Davis-Putnam

Comme son dérivé plus connu, l'algorithme DPLL, l'algorithme de Davis-Putnam utilise la propagation unitaire et l'élimination des littéraux purs. Mais l'appel récursif utilisé dans l'algorithme DPLL est remplacé par l'utilisation exhaustive de la règle de résolution sur une variable. L'algorithme de Davis-Putnam permet de prouver qu'un ensemble de clauses est satisfiable (ou non), mais contrairement à l'algorithme DPLL, il ne donne pas directement une affectation satisfaisant cet ensemble de clauses.

La résolution exhaustive pour une variable n'est appliquée que lorsque les deux autres (propagation unitaire et élimination des littéraux purs) ne sont plus possibles. Dans ce cas on choisit une variable (dison x) et on applique la règle de résolution sur toutes les clauses utilisant x : à partir d'une clause $A \vee x$ et d'une clause $B \vee \overline{x}$, on génère la clause $A \vee B$ (si celle-ci n'est pas une tautologie, c'est-à-dire ne comprend pas à la fois les littéraux y et $\overline{y}$). Ensuite, on supprime de l'ensemble de clauses toutes les clauses utilisant la variable x.

Exemple. On part des clauses initiales $C_1 = \overline{a} \vee c$, $C_2 = a \vee c$, $C_3 = b \vee \overline{c}$, $C_4 = a \vee \overline{b} \vee c$, $C_5 = \overline{b} \vee \overline{c}$, $C_6 = \overline{a} \vee \overline{b} \vee \overline{c}$, $C_7 = a \vee \overline{d}$ et $C_8 = b \vee \overline{d}$. On voit tout de suite que $\overline{d}$ est un littéral pur, on peut donc supprimer C_7 et C_8 de l'ensemble des clauses. Ensuite, on doit appliquer l'étape de résolution, sur a par exemple, ce qui donne les nouvelles clauses $C_{1,2} = c$ et $C_{1,4} = \overline{b} \vee c$ ($C_{2,6}$ et $C_{4,6}$ étant des tautologies) et permet de supprimer les clauses C_1, C_2, C_4, C_6. La clause $C_{1,2}$ est unitaire, permet de supprimer $C_{1,4}$ et de transformer C_3 et C_5 en $C'_3 = b$ et $C'_5 = \overline{b}$. Enfin, l'étape de résolution appliqué sur b (dernière variable

restante) crée une clause contradictoire, ce qui montre que l'ensemble de clauses initiale est insatisfiable.

On souhaire ici écrire la fonction

```
exhaustiveResolution :: (Ord a) => Formula a -> Formula a
```

du module **Formula** qui à partir d'une formule f détermine un littéral, dison x, et retourne une nouvelle formule f′ obtenue par résolution exhaustive de x.

(a) Écrire la fonction

```
anyLiteral :: (Eq a) => Formula a -> L.Literal a
```

du module **Formula** qui retourne un littéral quelconque apparaissant dans la formule. On suppose ici que la formule ne contient aucune clause unitaire et aucun littéral pur. (Vous remarquerez donc que n'importe quelle variable apparaît à la fois positivement et négativement dans la formule.)

(b) Écrire la fonction

```
clausesWithLiteral :: (Eq a) => Formula a -> L.Literal a -> [C.Clause a]
```

du module **Formula** qui retourne la liste des clauses de la formule qui contiennent le litéral passé en argument.

(c) Écrire la fonction

```
exhaustiveResolution :: (Ord a) => Formula a -> Formula a
```

du module **Formula** qui à partir d'une formule f détermine un littéral, disons x, et retourne une nouvelle formule f′ obtenue par résolution exhaustive de x.

(d) Écrire la fonction

```
isSolvable :: (Ord a) => Formula a -> Bool
```

du module **SatSolver** qui utilise l'algorithme de Davis-Putnam pour déterminer si une formule est satisfaisable. (On rappelle que l'algorithme de Davis-Purnam ne donne pas directement une affectation satisfaisant de cet ensemble de clauses.)

Appendix

2.1 Utilisation des différents modules

2.2 ghci

```
>>> import qualified Variable as V
>>> import qualified Literal as L
>>> import qualified Clause as C
>>> import qualified Formula as F
>>>
>>> let riri = V.Variable "riri"
>>> let fifi = V.Variable "fifi"
>>> let loulou = V.Variable "loulou"
>>>
>>> let priri = L.Positive riri      -- positive riri
>>> let nriri = L.Negative riri      -- negative riri
>>> let pfifi = L.Positive fifi      -- positive fifi
>>> let nfifi = L.Negative fifi      -- negative fifi
>>> let ploulou = L.Positive loulou  -- positive loulou
>>> let nloulou = L.Negative loulou  -- negative loulou
>>>
>>> let c1 = C.makeClause [priri,pfifi,ploulou]
>>> let c2 = C.makeClause [nriri,nfifi,nloulou]
>>> let c3 = C.makeClause [priri,nfifi]
>>> let c4 = C.makeClause [nriri,pfifi]
>>> let c5 = C.makeClause [priri,nfifi,nloulou]
>>>
>>> let f = F.makeFormula_ [c1,c2,c3,c4]
>>> f
Formula [Clause [Positive (Variable "fifi")
                ,Positive (Variable "loulou")
                ,Positive (Variable "riri")]
        ,Clause [Negative (Variable "fifi")
                ,Negative (Variable "loulou")
                ,Negative (Variable "riri")]
        ,Clause [Negative (Variable "fifi")
                ,Positive (Variable "riri")]
        ,Clause [Negative (Variable "riri")
                ,Positive (Variable "fifi")]
        ,Clause [Negative (Variable "fifi")
                ,Negative (Variable "loulou")
                ,Positive (Variable "riri")]]
>>>
>>> import SatSolver
>>> solve f
Just (fromList [(Variable "fifi",True)
               ,(Variable "loulou",False)
               ,(Variable "riri",True)])
>>>
```