Projet IR3
Sat Solver

S. Vialette

25 octobre 2019

Préambule

Le but de ce projet est de développer en Haskell une librairie permettant de manipuler des formules booléennes et un solveur simple. Résoudre des formules booléennes est un problème très important, tout autant pour l'informatique théorique que pour les très nombreuses applications que l'on peut modéliser (consulter par exemple `http://www.satcompetition.org/`).

1 Introduction

1.1 Formules de la logique propositionnelle

Les formules de la *logique propositionnelle* sont construites à partir de *variables propositionnelles* et des *connecteurs booléens* "et" ($\wedge$), "ou" ($\vee$) et "non" ($\neg$).

P	Q	$P \wedge Q$
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

P	Q	$P \vee Q$
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

P	$\neg P$
vrai	faux
faux	vrai

Une formule est *satisfaisable* s'il existe une assignation des variables propositionnelles qui rend la formule logiquement vraie [1].

Par exemple :

— La formule $(p \wedge q) \vee \neg p$ est satisfaisable car si p prend la valeur faux, la formule est évaluée à vrai ;

— La formule $(p \wedge \neg p)$ n'est pas satisfaisable car aucune valeur de p ne peut rendre la formule vraie.

Les règles de simplification peuvent être consultées sur le lien `http://sandbox.mc.edu/~bennet/cs110/boolalg/rules.html`, et des exemples de simplifications de formules sont visibles sur le lien `http://sandbox.mc.edu/~bennet/cs110/boolalg/simple.html`.

Dans ce projet, nous considérerons les formules booléennes munies de plus des connecteurs booléens "implication" ($\Rightarrow$), "équivalence" ($\Leftrightarrow$) et "ou exclusif" ($\oplus$).

P	Q	$P \Rightarrow Q$
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	vrai
faux	faux	vrai

P	Q	$P \Leftrightarrow Q$
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	vrai

P	Q	$P \oplus Q$
vrai	vrai	faux
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

1. Consulter par exemple l'entrée `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem` sur wikipedia.

1.2 Clauses et forme normale conjonctives

Un *littéral* ℓ est une variable propositionnelle v_j (*littéral positif*) ou la négation d'une variable propositionnelle $\neg v_j$. Une *clause* est une disjonction de la forme $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$, où les ℓ_i sont des littéraux. Une clause est *unitaire* si elle contient un seul litéral. Une formule du calcul propositionnel est en *forme normale conjonctive* (ou CNF sigle de l'anglais Conjunctive Normal Form) si elle est une conjonction de clauses.

Par exemple, soient $V = \{x_1, x_2, x_3\}$ un ensemble de variables propositionnelles et $f = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg v_2 \vee \neg x_3)$. Les formules $(x_1 \vee x_2)$, $(\neg x_1 \vee \neg x_2 \vee x_3)$ et $(x_1 \vee \neg v_2 \vee \neg x_3)$ sont des clauses. Leur conjonction f est une forme normale conjonctive.

Si la formule propositionnelle n'est pas sous forme normale conjonctive, on peut la transformer en une forme normale conjonctive équivalente de taille au plus exponentielle en la formule initiale. On peut aussi la transformer en une forme normale conjonctive équisatisfiable de taille linéaire en la formule initiale. Un algorithme de conversion est donné, par exemple, sur le lien `https://www.cs.jhu.edu/~jason/tutorials/convert-to-CNF.html`.

2 Formules

Il s'agit dans cette première partie de développer un ensemble de modules permettant de créer, simplifier et manipuler des formules. On trouve dans **`Data.Algorithm.Sat.Fml.Examples`** quelques formules que l'on doit pour voir créer (page 5).

Les types définis dans **`Data.Algorithm.Sat.Var`** (page 7) et **`Data.Algorithm.Sat.Lit`** (page 7)) sont imposés et donnés complet (vous pouvez bien sût ajouter d'autres fonctions si nécessaire).

Un module **`Data.Algorithm.Sat.Fml`** doit fournir le type des formules. Vous devez impérativement utiliser ce type.

```
import qualified Data.Algorithm.Sat.Var as Var

data Fml a = Or    (Fml a) (Fml a)
           | And   (Fml a) (Fml a)
           | Not   (Fml a)
           | Imply (Fml a) (Fml a)
           | Equiv (Fml a) (Fml a)
           | XOr   (Fml a) (Fml a)
           | Final (Var.Var a)
           deriving (Show, Eq, Ord)
```

Le module **`Data.Algorithm.Sat.Fml`** doit impérativement fournir - dans un premier temps - les fonctions suivantes :

— `toCNF :: Fml a -> Fml a` qui convertit une formule en une formule équivalent en forme normale conjonctive.

— `vars :: Fml a -> [Var.Var a]` qui retourne la liste des variables qui apparaissent dans une formule.

— `prettyPrinter :: Fml a -> String` qui propose une version lisible d'une formule.

Le module **`Data.Algorithm.Sat.Fml.Model`** propose des fonctions qui facilite l'écriture de formule :

— `atLeast :: (Eq t, Num t, Ord a) => t -> [Fml.Fml a] -> Fml.Fml a` qui prend pour arguments un entier k une liste de formules $[f_1, f_2, \ldots, f_n]$ et retourne la formule qui est satisfaisable lorsque au moins k des formules $f_1, f_2, \ldots, f_n$ sont satisfaisables.

— `anyOf :: (Ord a) => [Fml.Fml a] -> Fml.Fml a` qui prend pour argument une liste de formules $[f_1, f_2, \ldots, f_n]$ et retourne la formule qui est satisfaisable lorsque au moins une des formules $f_1, f_2, \ldots, f_n$ est satisfaisable.

— `noneOf :: (Ord a) => [Fml.Fml a] -> Fml.Fml a` qui prend pour argument une liste de formules $[f_1, f_2, \ldots, f_n]$ et retourne la formule qui est satisfaisable lorsque aucune des formules $f_1, f_2, \ldots, f_n$ n'est satisfaisable.

— `allOf :: (Ord a) => [Fml.Fml a] -> Fml.Fml a` qui prend pour argument une liste de formules $[f_1, f_2, \ldots, f_n]$ et retourne la formule qui est satisfaisable lorsque toutes les formules $f_1, f_2, \ldots, f_n$ sont satisfaisables.
— `exactlyOneOf :: (Ord a) => [Fml.Fml a] -> Fml.Fml a` qui prend pour argument une liste de formules $[f_1, f_2, \ldots, f_n]$ et retourne la formule qui est satisfaisable lorsque exactement une des formules $f_1, f_2, \ldots, f_n$ est satisfaisable.

3 Résoudre

Écrire un solveur de formule booléenne. Le problème est intrinsèquement difficile et l'approche la plus simple est d'utiliser une recherche arborescente à partir de la formule sous forme normale conjonctive. Pour faciliter l'écriture du solveur nous allons utiliser un type spécifique pour représenter les formules sous forme normale conjonctive.

3.1 Data.Algorithm.Sat.Solver.Clause

Le module `Data.Algorithm.Sat.CNFFml.Clause` permet de représenter les clauses sous une forme qui rend le traitement plus aisé. Le module introduit le type suivant :

```
import qualified Data.Algorithm.Sat.Lit        as Lit

-- A clause is a list of literals.
newtype Clause a = Clause { getLits :: [Lit.Lit a] } deriving (Eq)

instance (Show a) => Show (Clause a) where
  show Clause { getLits = ls } = "(" ++ L.intercalate "," (L.map show ls) ++ ")"
```

3.2 Data.Algorithm.Sat.Solver.CNFFml

Le module `Data.Algorithm.Sat.Solver.CNFFml` permet de représenter des ensembles de clauses (et donc des formules sous forme normale conjonctive). Le module introduit le type suivant :

```
import qualified Data.Algorithm.Sat.Solver.Clause as Solver.Clause

newtype CNFFml a = CNFFml { getClauses :: [Solver.Clause.Clause a] }
```

3.3 Data.Algorithm.Sat.Assignment

Pour représenter et manipuler une assignation des variables propositionnelles, nous utiliserons le type suivant (défini dans le module `Data.Algorithm.Sat.Assignment`) :

```
import qualified Data.Map.Strict as M
import qualified Data.Tuple      as T

import qualified Data.Algorithm.Sat.Lit as Lit
import qualified Data.Algorithm.Sat.Var as Var

newtype Assignment a = Assignment { getMap :: M.Map (Var.Var a) Bool }

instance (Show a, Ord a) => Show (Assignment a) where
  show = show . L.sort . L.map (Arrow.first Var.getName) . M.toList . getMap

-- |'empty' return the empty assignment.
mkEmpty :: Assignment a
mkEmpty = Assignment { getMap = M.empty }
```

```haskell
-- |'lookup' @v@ @m@ returns the boolean value asociated to variable @v@ in
-- the asignement @m@ (if it exists).
lookup :: (Ord a) => Var.Var a -> Assignment a -> Maybe Bool
lookup v = M.lookup v . getMap

...

-- |'insert' @l@ @m@ inserts literal @l@ in the assignment @m@ producing a new
-- assigment.
insert :: (Ord a) => Lit.Lit a -> Assignment a -> Assignment a
insert (Lit.F v) = Assignment . M.insert v False . getMap
insert (Lit.T v) = Assignment . M.insert v True  . getMap
```

3.4 Data.Algorithm.Sat.Solver

Le module **Data.Algorithm.Sat.Solver** implémente un algorithme de résolution. Il propose la fonction `solve :: (Ord a) => Fml.Fml a -> Maybe (Assignment.Assignment a)` qui permet de calculer une assignation des variables propositionnelles qui rend la formule logiquement vraie (si une telle asignation existe).

On utiliseras l'algorithme suivant pour une formule f.

1. Passer f sous forme normale conjonctive.
2. Si f contient une clause unitaire, soit ℓ le litéral de cette clause. Sinon, choisir un litéral ℓ qui référence une variable ayant le plus grand nombre d'occurrences dans f.
   — Soit f′ la formule obtenue en simplifiant la formule f par ℓ. Si f′ est satisfaisable, retourner une assignation des variables propositionnelles qui rend la formule logiquement vraie.
   — Soit f″ la formule obtenue en simplifiant la formule f par $\neg\ell$. Si f″ est satisfaisable, retourner une assignation des variables propositionnelles qui rend la formule logiquement vraie.
   — La formule f n'est pas satisfaisable.

3.5 Data.Algorithm.Sat.Query

Le module **Data.Algorithm.Sat.Query** propose plusieurs fonctions pour faciliter l'utilisation du solveur.

— `satisfiable :: (Ord a) => Fml.Fml a -> Bool` qui décide si une formule booléenne est satisfaisable.
— `satisfyingAssignment :: (Ord a) => Fml.Fml a -> Maybe (Assignment.Assignment a)` qui retourne une assignation des variables propositionnelles qui rend la formule logiquement vraie (si une telle assignation existe).
— `satisfyingAssignments :: (Ord a) => Fml.Fml a -> [Assignment.Assignment a]` qui retourne toutes les assignations des variables propositionnelles qui rendent la formule logiquement vraie.
— `tautology :: (Ord a) => Fml.Fml a -> Bool` qui décide si toutes les assignations des variables propositionnelles rendent la formule logiquement vraie.

4 Annexes

4.1 Data.Algorithm.Sat.Fml.Examples

```
module Data.Algorithm.Sat.Fml.Examples
  (
  fml1
, fml2
, fml3
, fml4
, fml5
, fml6
, fml7
, fml8
, fml9
, fml10
, fml11
, fml12
, fml13
  ) where

import qualified Data.Algorithm.Sat.Fml       as Fml
import qualified Data.Algorithm.Sat.Fml.Model as Fml.Model

vA = Fml.mkVar 'A'
vB = Fml.mkVar 'B'
vC = Fml.mkVar 'C'
vD = Fml.mkVar 'D'
vE = Fml.mkVar 'E'
vF = Fml.mkVar 'F'

-- C + -(BC)
fml1 :: Fml.Fml Char
fml1 = Fml.Or vC (Fml.Not (Fml.And vB vC))

-- (-A)(-B)(-A + B)(-B + B)
fml2 :: Fml.Fml Char
fml2 = Fml.multAnd [f, g, h]
  where
    f = Fml.And (Fml.Not vA) (Fml.Not vB)
    g = Fml.Or (Fml.Not vA) vB
    h = Fml.Or (Fml.Not vB) vB

-- (A + C)(AD + A(-D)) + AC + C
fml3 :: Fml.Fml Char
fml3 = Fml.multOr [f, g, h]
  where
    f = Fml.And (Fml.Or vA vC) (Fml.Or (Fml.And vA vD) (Fml.And vA (Fml.Not vD)))
    g = Fml.And vA vC
    h = vC

-- (-A)(A + B) + (B + AA)(A + (-B)):
fml4 :: Fml.Fml Char
fml4 = Fml.Or f g
  where
    f = Fml.And (Fml.Not vA) (Fml.Or vA vB)
```

```
    g = Fml.And (Fml.Or vB (Fml.And vA vA)) (Fml.Or vA (Fml.Not vB))

-- (A => (B + C)((-C) + D)) + (BC => (-A)(-D))
fml5 :: Fml.Fml Char
fml5 = Fml.Or f g
  where
    f = Fml.Imply vA (Fml.And (Fml.Or vB vC) (Fml.Or (Fml.Not vC) vD))
    g = Fml.Imply (Fml.And vB vC) (Fml.And (Fml.Not vA) (Fml.Not vD))

-- -(-(A)) + -(-(-B))
fml6 :: Fml.Fml Char
fml6 = Fml.Or f g
  where
    f = Fml.Not (Fml.Not vA)
    g = Fml.Not (Fml.Not (Fml.Not vB))

-- (AB + CD + EF) <=> (AB => (CD <=> EF))
fml7 :: Fml.Fml Char
fml7 = Fml.Equiv f g
  where
    f = Fml.multOr [Fml.And vA vB, Fml.And vC vD, Fml.And vE vF]
    g = Fml.Imply (Fml.And vA vB) (Fml.Equiv (Fml.And vC vD) (Fml.And vE vF))

-- (A+B) (A+(-B)) ((-A)+B) ((-A)+(-B))
fml8 :: Fml.Fml Char
fml8 = Fml.multAnd [f, g, h, i]
  where
    f = Fml.Or vA vB
    g = Fml.Or vA (Fml.Not vB)
    h = Fml.Or (Fml.Not vA) vB
    i = Fml.Or (Fml.Not vA) (Fml.Not vB)

-- (A(-B) + EF) (AB => ((-B)C + B(-C) + (-F)) (AB <=> C(-D)) (AB => (CD + E(-F)))
fml9 :: Fml.Fml Char
fml9 = Fml.multAnd [f, g, h, i]
  where
    f = Fml.Or (Fml.And vA (Fml.Not vB)) (Fml.And vE vF)
    g = Fml.Imply (Fml.And vA vB) (Fml.multOr [Fml.And (Fml.Not vB) vC, Fml.And vB (Fml.
    h = Fml.Equiv (Fml.And vA vB) (Fml.And vC (Fml.Not vD))
    i = Fml.Imply (Fml.And vA vB) (Fml.Or (Fml.And vC vD) (Fml.And vE (Fml.Not vF)))

fml10 :: Fml.Fml Char
fml10 = Fml.multAnd [p1, p2, p3, p4]
  where
    p1 = Fml.Model.atLeast 3 [vA, vB, vC, vD, vE, vF]
    p2 = Fml.multOr [Fml.Not vA, Fml.Not vB, vC, vF]
    p3 = Fml.multOr [Fml.Not vB, Fml.Not vD, Fml.Not vE]
    p4 = Fml.Imply (Fml.Or vA vB) (Fml.And vC vE)

fml11 :: Fml.Fml Char
fml11 = Fml.And p1 p2
  where
    p1 = Fml.multOr [vA, vB, vC, vD, vE, vF]
    p2 = Fml.Model.noneOf [vA, vC, vE]

fml12 :: Fml.Fml Char
```

```haskell
fml12 = Fml.And p1 p2
  where
    p1 = Fml.And (Fml.Or (Fml.Not vA) (Fml.Not vB)) (Fml.Or vC (Fml.Not vB))
    p2 = Fml.Model.allOf [vA, vB]

fml13 :: Fml.Fml Char
fml13 = Fml.multAnd [p1, p2, p3, p4]
  where
    p1 = Fml.Model.exactlyOneOf [vA, vB, vC]
    p2 = Fml.And vA (Fml.Not vE)
    p3 = Fml.Imply (Fml.Model.allOf [vA, vB]) (Fml.Model.allOf [vC, vE])
    p4 = Fml.Equiv (Fml.And vA vC) (Fml.multAnd [vB, vE, vF])
```

4.2 Data.Algorithm.Sat.Var

```haskell
module Data.Algorithm.Sat.Var
  (
  -- * Type
  Var(..)

  -- * Making
, mk
  ) where

-- |@Var@ type definition.
newtype Var a = Var { getName :: a } deriving (Eq, Ord)

instance (Show a) => Show (Var a) where
  show Var { getName = n } = "Var " ++ show n

-- |'mk' @x@ makes the variable with name @x@.
--
-- >>> mk 'a'
-- Var 'a'
mk :: a -> Var a
mk = Var
```

4.3 Data.Algorithm.Sat.Lit

```haskell
module Data.Algorithm.Sat.Lit
  (
  -- * Type
  Lit(..)

  -- * Making
, mkFalse
, mkTrue

  -- * Querying
, getVar

  -- * Transforming
, neg
  ) where

import qualified Data.Algorithm.Sat.Var as Var
```

```haskell
-- |Literal type definition.
data Lit a = F (Var.Var a) | T (Var.Var a) deriving (Eq, Ord)

instance (Show a) => Show (Lit a) where
  show (F v) = "-" ++ show v
  show (T v) =        show v

-- |'mkFalse' @v@ returns the false literal on variable @v@.
--
-- >>> let v = Var.mk 'a' in mkFalse v
-- -Var 'a'
mkFalse :: Var.Var a -> Lit a
mkFalse = F

-- |'mkTrue' @v@ returns the true literal on variable @v@.
--
-- >>> let v = Var.mk 'a' in mkTrue v
-- Var 'a'
mkTrue :: Var.Var a -> Lit a
mkTrue = T

-- |'neg' @l@ negates litteral @v@.
--
-- >>> let v = Var.mk 'a' in neg (mkFalse v)
-- Var 'a'
-- >>> let v = Var.mk 'a' in neg (mkTrue v)
-- -Var 'a'
neg :: Lit a -> Lit a
neg (F v) = T v
neg (T v) = F v

-- |'getVar' @l@ returns the variable on which literal @l@ is defined.
--
-- >>> let v = Var.mk 'a' in getVar (mkFalse v)
-- Var 'a'
-- >>> let v = Var.mk 'a' in getVar (mkTrue v)
-- Var 'a'
getVar :: Lit a -> Var.Var a
getVar (F v) = v
getVar (T v) = v
```