**The project** aims to provide a Haskell implementation for manipulating logical formulas and converting them to Conjunctive Normal Form (CNF). It comprises three modules: `Formula.hs` for representing and operating on logical formulas, `Literal.hs` for handling literals, and `NormalForm.hs` for converting formulas to CNF and applying resolution rules. These modules offer various functions to perform basic logical operations, check formula properties, and simplify or evaluate formulas under given environments. The project facilitates the exploration of logical formula operations, paving the way for further work on automated theorem proving and logical analysis.

The NormalForm module in Haskell is designed to manipulate logical formulas and convert them into Conjunctive Normal Form (CNF). Below are the descriptions of the functions in the NormalForm module and the issues identified:

### Functions Description:

1. fromFormula Function:
   * Purpose: Convert a logical formula into CNF.
   * Input: A logical formula of type Formula.
   * Output: A CNF representation of type CNF.
   * Methods Used:
     + Elimination of Implications: eliminateImplication function handles this.
     + Conversion to Negation Normal Form (NNF): toNNF function handles this.
     + Distribution: distribute function handles this.
     + Conversion to CNF: toCNF function handles this.
2. toFormulaCNF Function:
   * Purpose: Convert CNF back to a logical formula.
   * Input: A CNF representation of type CNF.
   * Output: A logical formula of type Formula.
3. robinson Function:
   * Purpose: Implement Robinson's resolution rule on a set of clauses.
   * Input: A set of clauses of type Set (Set Literal).
   * Output: Possibly a new set of clauses of type Maybe (Set (Set Literal)).

### Identified Issue:

The primary issue appears to reside in the fromFormula function where the conversion from a logical formula to CNF is not functioning correctly, as per the test output shared. The incorrect CNF representation generated does not match the expected CNF for some input formulas.

The distribute function's logic seems to have a flaw. The distribution step in the conversion to CNF should apply the distributive law: A∧(B∨C)≡(A∧B)∨(A∧C) and vice versa, until all the ANDs and ORs are properly distributed, leading to a CNF. However, the implementation seems to be producing incorrect results for some input formulas.

Moreover, the toCNF function may also not be handling the clauses and literals correctly, especially in the case of OR operations, which seems to be contributing to the incorrect CNF conversion.

**Code module NormalForm:**

module NormalForm(

CNF(..),

size,

toFormulaCNF,

fromFormula,

robinson

) where

import Data.Set (Set)

import qualified Data.Set as Set

import Formula (Formula(..))

import Literal

-- | A conjunctive normal form

newtype CNF = CNF (Set (Set Literal))

deriving Eq

instance Show CNF where

show (CNF clauses) = unwords . map show . Set.toList $ clauses

-- | Size (number of literals)

size :: CNF -> Int

size (CNF clauses) = sum (Set.size <$> Set.toList clauses)

-- | Convert normal form to logical formula

toFormulaCNF :: CNF -> Formula

toFormulaCNF (CNF clauses) =

foldl1 And [foldl1 Or [Literal.toFormula lit | lit <- Set.toList clause] | clause <- Set.toList clauses]

-- | Convert logical formula to normal form

fromFormula :: Formula -> CNF

fromFormula formula =

let formulaNoImp = eliminateImplication formula

nnf = toNNF formulaNoImp

distributed = distribute nnf

in CNF (toCNF distributed)

-- | Eliminate implications recursively

eliminateImplication :: Formula -> Formula

eliminateImplication (Imp f g) = Or (Not f) g

eliminateImplication (And f1 f2) = And (eliminateImplication f1) (eliminateImplication f2)

eliminateImplication f = f

-- | Convert formula to NNF

toNNF :: Formula -> Formula

toNNF (Not (And f1 f2)) = Or (toNNF (Not f1)) (toNNF (Not f2))

toNNF (Not (Or f1 f2)) = And (toNNF (Not f1)) (toNNF (Not f2))

toNNF (Not (Not f)) = toNNF f

toNNF (And f1 f2) = And (toNNF f1) (toNNF f2)

toNNF (Or f1 f2) = Or (toNNF f1) (toNNF f2)

toNNF f = f

-- | Distribute And over Or

distribute :: Formula -> Formula

distribute (And f1 (Or f2 f3)) = Or (distribute (And f1 f2)) (distribute (And f1 f3))

distribute (And (Or f1 f2) f3) = Or (distribute (And f1 f3)) (distribute (And f2 f3))

distribute (And f1 f2) = And (distribute f1) (distribute f2)

distribute (Or f1 f2) = Or (distribute f1) (distribute f2)

distribute f = f -- Base case: return the formula if it's a literal or a negation

-- | Convert formula to CNF

toCNF :: Formula -> Set (Set Literal)

toCNF formula =

case formula of

Const b -> Set.singleton (Set.singleton (fromBool b))

Var s -> Set.singleton (Set.singleton (fromPositive s))

Not (Const b) -> Set.singleton (Set.singleton (fromBool (not b)))

Not (Var s) -> Set.singleton (Set.singleton (fromNegative s))

And f1 f2 -> Set.union (toCNF f1) (toCNF f2)

Or f1 f2 ->

let leftClauses = Set.toList (toCNF f1)

rightClauses = Set.toList (toCNF f2)

in Set.fromList [Set.fromList [l | lc <- [lClause, rClause], l <- Set.toList lc]

| lClause <- leftClauses, rClause <- rightClauses]

-- ... (reste du code pour les règles de Robinson)

-- | Apply ROBINSON's rule on clauses

robinson :: Set (Set Literal) -> Maybe (Set (Set Literal))

robinson clauses =

let pairs = [(c1, c2) | c1 <- Set.toList clauses, c2 <- Set.toList clauses, c1 /= c2] in

findResolvent pairs clauses

findResolvent :: [(Set Literal, Set Literal)] -> Set (Set Literal) -> Maybe (Set (Set Literal))

findResolvent [] \_ = Nothing

findResolvent ((c1, c2):rest) clauses =

case resolvent c1 c2 of

Just clause -> Just (Set.insert clause clauses)

Nothing -> findResolvent rest clauses

resolvent :: Set Literal -> Set Literal -> Maybe (Set Literal)

resolvent c1 c2 =

let lits = [l | l <- Set.toList c1, neg l `Set.member` c2] in

if null lits

then Nothing

else Just $ foldr Set.delete c1 lits `Set.union` foldr Set.delete c2 (map neg lits)

### Formula.hs Description:

This module defines the data structure and operations for handling logical formulas.

1. Data Types:
   * Formula: Represents a logical formula with constructors for constants (Const), variables (Var), negation (Not), conjunction (And), disjunction (Or), and implication (Imp).
2. Functions:
   * fromBool: Converts a Boolean value to a Formula of type Const.
   * fromString: Converts a string to a Formula of type Var.
   * neg, conj, disj, implies: Basic logical operations for negation, conjunction, disjunction, and implication, respectively.
   * isLiteral: Checks if a formula is a literal.
   * has: Checks if a formula contains a specific variable.
   * size: Computes the size of a formula in terms of the number of operators.
   * variables: Extracts the set of variables from a formula.
   * evaluate: Evaluates a formula under a given environment.
   * (<=>): Checks logical equivalence between two formulas.
   * tautology: Checks if a formula is a tautology.
   * simplify: Simplifies a formula by applying basic simplification rules.

**Code  Formula:**

module Formula(Formula(..), fromBool, fromString, neg, conj, disj, implies, isLiteral, has, size, variables, Environment, evaluate, (<=>), tautology, simplify) where

import Data.Kind

import Data.Set (Set)

import qualified Data.Set as Set

import Data.Map (Map)

import qualified Data.Map as Map

-- La structure principale de notre formule

data Formula

= Const Bool -- Une constante booléenne (True ou False)

| Var String -- Une variable logique

| Not Formula -- Négation

| And Formula Formula -- Conjonction (ET logique)

| Or Formula Formula -- Disjonction (OU logique)

| Imp Formula Formula -- Implication

deriving (Show, Eq)

-- \*\* Constructeurs de Formules \*\* --

fromBool :: Bool -> Formula

fromBool = Const

fromString :: String -> Formula

fromString = Var

-- \*\* Opérations de Base sur les Formules \*\* --

neg :: Formula -> Formula

neg = Not

conj :: Formula -> Formula -> Formula

conj = And

disj :: Formula -> Formula -> Formula

disj = Or

implies :: Formula -> Formula -> Formula

implies = Imp

-- \*\* Propriétés des Formules \*\* --

isLiteral :: Formula -> Bool

isLiteral (Var \_) = True

isLiteral (Not (Var \_)) = True

isLiteral \_ = False

has :: Formula -> String -> Bool

(Var v) `has` name = v == name

(Not f) `has` name = f `has` name

(And f1 f2) `has` name = f1 `has` name || f2 `has` name

(Or f1 f2) `has` name = f1 `has` name || f2 `has` name

(Imp f1 f2) `has` name = f1 `has` name || f2 `has` name

\_ `has` \_ = False

size :: Formula -> Int

size (Var \_) = 0

size (Not f) = 1 + size f

size (And f1 f2) = 1 + size f1 + size f2

size (Or f1 f2) = 1 + size f1 + size f2

size (Imp f1 f2) = 1 + size f1 + size f2

size (Const \_) = 0

variables :: Formula -> Set String

variables (Var v) = Set.singleton v

variables (Not f) = variables f

variables (And f1 f2) = Set.union (variables f1) (variables f2)

variables (Or f1 f2) = Set.union (variables f1) (variables f2)

variables (Imp f1 f2) = Set.union (variables f1) (variables f2)

variables (Const \_) = Set.empty

-- Type pour les environnements (associations variable/valeur)

type Environment = Map String Bool

-- Évaluation d'une formule dans un environnement donné

evaluate :: Environment -> Formula -> Maybe Bool

evaluate env (Const b) = Just b

evaluate env (Var v) = Map.lookup v env

evaluate env (Not f) = not <$> evaluate env f

evaluate env (And f1 f2) = do

b1 <- evaluate env f1

b2 <- evaluate env f2

return (b1 && b2)

evaluate env (Or f1 f2) = do

b1 <- evaluate env f1

b2 <- evaluate env f2

return (b1 || b2)

evaluate env (Imp f1 f2) = do

b1 <- evaluate env f1

b2 <- evaluate env f2

return (not b1 || b2)

-- Équivalence logique

(<=>) :: Formula -> Formula -> Bool

f <=> g = all (\env -> eval f env == eval g env) envs

where

vars = Set.union (variables f) (variables g)

envs = allEnvironments $ Set.toList vars

eval formula env = case evaluate (Map.fromList env) formula of

Just val -> val

Nothing -> error "Incomplete environment"

allEnvironments :: [String] -> [[(String, Bool)]]

allEnvironments [] = [[]]

allEnvironments (v:vs) =

[(v, True) : env | env <- allEnvironments vs] ++

[(v, False) : env | env <- allEnvironments vs]

-- Vérifier si la formule est une tautologie

tautology :: Formula -> Bool

tautology f = all (\env -> eval f env) envs

where

vars = variables f

envs = allEnvironments $ Set.toList vars

eval formula env = case evaluate (Map.fromList env) formula of

Just val -> val

Nothing -> error "Incomplete environment"

-- Simplification (cette version est rudimentaire)

simplify :: Formula -> Formula

simplify (And (Const True) f) = simplify f

simplify (And f (Const True)) = simplify f

simplify (And (Const False) \_) = Const False

simplify (And \_ (Const False)) = Const False

simplify (Or (Const True) \_) = Const True

simplify (Or \_ (Const True)) = Const True

simplify (Or (Const False) f) = simplify f

simplify (Or f (Const False)) = simplify f

simplify (Not (Const True)) = Const False

simplify (Not (Const False)) = Const True

simplify (Not (Not f)) = simplify f

simplify (Imp f1 f2) = simplify (Or (Not f1) f2)

simplify f = f -- Pour les autres cas, on ne simplifie pas

### **Literal.hs Description:**

This module defines the data structure and operations for handling literals, which are the basic elements in a normal form.

1. Data Types:
   * Literal: Represents a literal with constructors for constants (ConstLit), positive literals (PositiveLit), and negative literals (NegativeLit).
2. Functions:
   * fromBool: Converts a Boolean value to a Literal of type ConstLit.
   * fromPositive: Converts a string to a Literal of type PositiveLit.
   * fromNegative: Converts a string to a Literal of type NegativeLit.
   * neg: Negates a literal, switching between positive and negative literals, or negating a constant literal.
   * toFormula: Converts a Literal to a Formula based on its type.

**Code module Literal :**

module Literal(Literal(..), fromBool, fromPositive, fromNegative, neg, toFormula) where

import Data.Kind

import qualified Formula

data Literal

= ConstLit Bool

| PositiveLit String

| NegativeLit String

deriving (Eq, Ord)

instance Show Literal where

show (ConstLit b) = show b

show (PositiveLit s) = s

show (NegativeLit s) = "¬" ++ s

fromBool :: Bool -> Literal

fromBool = ConstLit

fromPositive :: String -> Literal

fromPositive = PositiveLit

fromNegative :: String -> Literal

fromNegative = NegativeLit

neg :: Literal -> Literal

neg (ConstLit b) = ConstLit (not b)

neg (PositiveLit s) = NegativeLit s

neg (NegativeLit s) = PositiveLit s

toFormula :: Literal -> Formula.Formula

toFormula (ConstLit b) = Formula.Const b

toFormula (PositiveLit s) = Formula.Var s

toFormula (NegativeLit s) = Formula.Not (Formula.Var s)