Autres Paradigmes

DM Haskell

HUET Bryan
21701042

SAHIN Tolga
21801042

Année universitaire 2020

Table des matières

1 Mise sous forme clausale d'une formule du calcul propositionnel (Partie 1)

Pour la bonne compréhension du dm, nous rappellerons à quel opérateurs logique correspond chaque caractère :
Soit g et d deux formules,
g̃ : Non g
g & d : g Et d
g v d : g Ou d
g => d : g Implique d
g <=> d : g Équivaut d

1.1 Visualiser une formule : Question 1

visuFormule

```
visuFormule : : Formule -> String

visuFormule (Var p) = p
visuFormule (Non f) = "~" ++ visuFormule f
visuFormule (Et g d) = "(" ++ (visuFormule g) ++ " & "
++ (visuFormule d) ++ ")"
visuFormule (Ou g d) = "(" ++ (visuFormule g) ++ " v "
++ (visuFormule d) ++ ")"
visuFormule (Imp g d) = "(" ++ (visuFormule g) ++ " => "
++ (visuFormule d) ++ ")"
visuFormule (Equi g d) ="(" ++ (visuFormule g) ++ " <=> "
++ (visuFormule d) ++ ")"
```

1.2 Faire disparaître des opérateurs : Question 2 & 3

Soit g et d deux formule,
- Il est possible de remplacer l'opérateur d'implication dans g => d par :
(g => d) = (g̃ v d)
- Il est possible de remplacer l'opérateur d'équivalence dans g <=> d par :
(g <=> d) = ((g => d) & (d => g))

elimine

```
elimine : : Formule -> Formule

elimine (Var p) = (Var p)
elimine (Non f) = (Non (elimine f))
elimine (Et g d) = (Et (elimine g) (elimine d))
elimine (Ou g d) =(Ou (elimine g) (elimine d))
elimine (Imp g d) = (Ou (Non (elimine g)) (elimine d))
elimine (Equi g d) = (Et (elimine (Imp (elimine g) (elimine d)))
(elimine (Imp (elimine d) (elimine g))))
```

1.3 Amener les négations devant les littéraux positifs : Question 4,5,6

Soit g, d et f des formules,
Grâce à la loi de la double négation, Non(Non f) devient f.
Les deux lois de Morgan sont les suivantes :
- Non (g v d) <=> ($\tilde{g}$) & ($\tilde{d}$)
- Non (g & d) <=> ($\tilde{g}$) v ($\tilde{d}$)

disNon doit appliquer les lois de Morgan et de la double négation.

ameneNon, disNon

```
ameneNon, disNon : : Formule -> Formule

ameneNon (Var p) = (Var p)
ameneNon (Non f) = disNon f
ameneNon (Et g d) = (Et (ameneNon g) (ameneNon d))
ameneNon (Ou g d) = (Ou (ameneNon g) (ameneNon d))

disNon (Var p) = (Non (Var p))
disNon (Non f) = (f)
disNon (Et g d) = (Ou (disNon g) (disNon d))
disNon (Ou g d) = (Et (disNon g) (disNon d))
```

1.4 Faire apparaître une conjonction de clauses : Question 7 & 8

developper

```
developper : : Formule -> Formule -> Formule

developper (Et g d) x = (Et (Ou x g) (Ou x d))
developper x (Et g d) = (Et (Ou x g) (Ou x d))
developper x y = (Ou x y)
```

formeClausale

```
formeClausale : : Formule -> Formule

formeClausale f = normalise (ameneNon (elimine f))
```

2 Résolvante et principe de résolution (Partie 2)

2.1 Transformer une Formule en une FormuleBis : Question 9

etToListe

```
etToListe :: Formule -> FormuleBis

etToListe (Et g d) = (ouToListe g) : (etToListe d)
etToListe f = [ouToListe f]
```

ouToListe

```
ouToListe :: Formule -> Clause

ouToListe (Ou g d) = (ouToListe g )++(ouToListe d)
ouToListe f = [f]
```

2.2 Résolvante de deux clauses : Question 10,11,12

neg

```
neg :: Formule -> Formule

neg (Non f) = f
neg f = (Non f)
```

sontLiees

```
sontLiees :: Clause -> Clause -> Bool

sontLiees [] _ = False
sontLiees (x :xs) ys = ((neg x) `elem` ys) || (sontLiees xs ys)
```

resolvante

```
resolvante : : Clause -> Clause -> Clause

resolvante [] ys = []
resolvante (x :xs) (y :ys)
    | ((neg x) `elem` (y :ys))== True = xs ++ (delete (neg x) (y :ys))
    | ((neg y) `elem` (x :xs))== True = ys ++ (delete (neg y) (x :xs))
    | otherwise = [x] ++ [y] ++ resolvante xs ys
```

3 Application : les sorites de Lewis Carroll

3.1 Les logiciens

1. Tous les individus sains d'esprit sont de possibles logiciens ;
2. Aucun malade mental n'est un juré possible ;
3. Aucun de vos enfants n'est un logicien possible ;
4. Les non malade mental sont sains d'esprit ;

Donc ...

On considère les propositions suivantes :
- saint : être saint d'esprit
- logicien : être logicien
- malade : être malade mental
- enfant : être un enfant

On modélise cette sorite sous forme d'une formule :
1 –> saint => logicien
2 –> malade => Non logicien
3 –> enfant => Non logicien
4 –> Non malade => saint

Traduction Haskell

```
logicien = (Et (Imp (Var "saint") (Var ("logicien")))
              (Et (Imp (Var "malade") (Non(Var "saint")))
                (Et (Imp (Var "enfant") (Non(Var "logicien")))
                  (Imp (Non (Var "malade")) (Var "saint")))))

> deduire logicien
[Non (Var "saint"),Non (Var "enfant")]
```

3.2 L'école

1. Aucun enfant de moins de douze ans dans cette école n'est interne ;
2. Tous les enfants studieux ont les cheveux roux ;
3. Aucun des externes n'est helléniste ;
4. Seuls les élèves de moins de douze ans sont paresseux ;
5. Les non externes sont internes ;
6. Les non paresseux sont studieux ;

Donc ...

On considère les propositions suivantes :
- douze : enfant de moins de douze ans
- interne : être interne
- externe : être externe
- roux : avoir les cheveux roux
- helléniste : être helléniste
- studieux : être studieux
- paresseux : être paresseux

On modélise cette sorite sous forme d'une formule :
1 −> douze => Non interne
2 −> studieux => roux
3 −> externe => Non helléniste
4 −> paresseux => douze
5 −> Non externe <=> interne 6 −> Non paresseux <=> studieux

Traduction Haskell

```
ecole = (Et (Imp (Var "douze") (Non (Var "interne")))
          (Et (Imp (Var "studieux") (Var "roux"))
           (Et (Imp (Var "externe") (Non (Var "helléniste")))
            (Et (Imp (Var "paresseux") (Var "douze"))
             (Et (Imp (Non (Var "externe")) (Var "interne"))
              (Imp (Non (Var "paresseux")) (Var "studieux")))))))
```

```
> deduire ecole
[Non (Var "helleniste"),Var "roux"]
```