

Programmation Fonctionnelle en Haskell

Chapitre 5 Applications en Logique

*Licence « Sciences et Technologies »
Parcours « Informatique » – 2ème année*

UFR Sciences – Université de Caen

Patrice Enjalbert

Département d'Informatique

Patrice.Enjalbert@info.unicaen.fr

Présentation

- Logique vue du point de vue de **l'Intelligence Artificielle** : **représentation de connaissance** et **raisonnement automatique** sur ces connaissances
Utilisera des notions vues en « maths », dans une visée plus « pratique »
- Plan
 1. **Logique (propositionnelle)** : rappels.
 2. **Codage de formules en Haskell**. Quelques manipulations utiles. Poursuivi en TD.
 3. **Une méthode de décision** : substitutions
 4. **Rappel: Formes Normales Conjonctive et Disjonctive**

1. Logique propositionnelle

1. Syntaxe

Soit V un ensemble de *variables propositionnelles* (notées ici: $p, q, r, \dots$)

L'ensemble des *formules* du Calcul Propositionnel est défini inductivement.

- (i) Toute variable propositionnelle est une formule.
- (ii) Si A et B sont des formules $A \wedge B$, $A \vee B$, $A \rightarrow B$ et $\neg A$ sont des formules.
- (iii) Toute formule est obtenue en appliquant les règles (i) et (ii) un nombre fini de fois.

$A \wedge B$ se lit "A et B". *conjonction* de A et B.

$A \vee B$ se lit "A ou B". *disjonction* de A et B.

$A \rightarrow B$ se lit "A implique B". *implication* .

$\neg A$ se lit "non A". *négation* .

$\wedge, \vee, \rightarrow, \neg$ sont des *connecteurs logiques*.

Exemple : p $p \wedge q$ $(p \wedge q) \vee (\neg r)$ $((p \wedge q) \vee (\neg r)) \rightarrow (p \vee (\neg r))$.

- **Remarques**

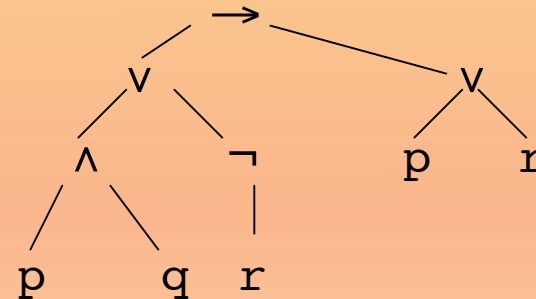
- **Règles de priorité.** En général on considérera que $\neg$ est prioritaire sur $\wedge$ et $\vee$, lesquels sont prioritaires sur $\rightarrow$.

$((((p \wedge q) \vee (\neg r)) \rightarrow (p \vee (\neg r))))$ peut s'écrire $p \wedge q \vee \neg r \rightarrow p \vee \neg r$

- **Représentation comme arbre** (comme toute expression « formelle »)

Exemple

$(p \wedge q) \vee \neg r \rightarrow p \vee r$



- **Intuitivement :**

- Une variable code une **propriété élémentaire** du « système étudié » : « *il fait beau* », « *la porte est ouverte* », « *le voyant b45c est allumé* »...
- Une formule code une **propriété complexe** du système : « *Si il fait beau alors je vais à la plage* », « *si le voyant b45c est allumé alors pb type a ou pb type b* »

2. Vérité

- Valuation

Assignment de valeurs de vérité à chaque variable.

Ex: $v0 = \{p \rightarrow 0, q \rightarrow 1, r \rightarrow 0\}$ $v1 = \{p \rightarrow 1, q \rightarrow 1, r \rightarrow 0\}$

- Valeur d'une formule pour une valuation

Pour une valuation v donnée, on peut évaluer la formule F (tables de vérité des connecteurs, cf cours de maths)

$Val(v, F)$ dans $\{0, 1\}$

Ex. $F0 = p \wedge (q \vee r)$ $val(v0, F0) = 0$ $val(v1, F0) = 1$

- Satisfaction d'une formule pour (par) une valuation

Def : v satisfait F (F vraie pour $v...$) si $val(v, F) = 1$

- Intuitivement :

- Valuation = un état du « système étudié »

$v3 = \{fait_beau \rightarrow 1, porte_ouverte \rightarrow 0, voyant\ b45_allumé \rightarrow 0...\}$

- v satisfait F si F est vraie dans l'état du système décrit par v

$v3$ satisfait $(fait_beau \vee porte_ouverte) \wedge \neg b45_allumé$

- Satisfiabilité

Definition

- F (formule) est satisfiable si il existe v qui satisfait F
- $\{F_1, F_2, \dots, F_n\}$ satisfiable si il existe v qui satisfait tous le F_i

Exemples :

- $\{p, \neg p\}$ non satisfiable (= insatisfiable, ou "insat")
- $\{p \rightarrow q, p, \neg q\}$ non satisfiable (insat)
- $\{p \rightarrow q \vee r, p, \neg q\}$ satisfiable (sat)

- Conséquence logique

- Definition $H = \{H_1 \dots H_n\}$ à pour conséquence A
Hypothèses Conclusion
- ssi toute valuation v qui satisfait H , satisfait aussi A
 A est vraie dès que H l'est

Propriété : *Conséquence logique* peut se ramener à *Satisfiabilité*

ssi $\{H1...Hn\}$ à pour conséquence A
 $\{H1...Hn, \neg A\}$ insatisfiable

Exemples :

$\{p \rightarrow q, p\}$ à pour conséquence q car $\{p \rightarrow q, p, \neg q\}$ insat
 $\{p \vee (r \wedge q), \neg r\}$ à pour conséquence p car $\{p \vee (r \wedge q), \neg r, \neg p\}$ insat

3. Application : formalisation de problèmes en logique

Soit « « « l'analyse politique » » » suivantes :

- (1) Si la gauche gagne les régionales, si le gouvernement s'engage dans une politique de rigueur on aura une crise sociale
- (2) Si le PS et les écolos s'unissent, la gauche sera crédible, et seulement dans ce cas
- (3) S'il y a une crise sociale et que la gauche est crédible, la droite perdra les élections de 2012

On veut savoir :

Pb1 : Si cet ensemble d'analyses est cohérent

Pb2 : Si elles impliquent que la Droite va perdre les élections si elle pratique la rigueur et la gauche gagne les régionales

Pb3 : Si ce n'est pas exact, quelle(s) autre(s) conditions sont nécessaires pour que la gauche gagne

- (1) Si la gauche gagne les régionales, si le gouvernement s'engage dans une politique de rigueur on aura une crise sociale
- (2) Si le PS et les écologistes s'unissent, la gauche sera crédible
- (3) S'il y a une crise sociale et que la gauche est crédible, la droite perdra les élections de 2012

Formalisation et résolution

Etape 1. Choisir des variables pour symboliser les « faits » mentionnés dans les énoncés et les coder en logique

Exemple : g_gagne_region pour « la gauche gagne », $rigueur$ pour « le gouvernement pratique la rigueur » etc.

- (G1) $g_gagne_region \rightarrow (rigueur \rightarrow crise)$
- (G2) $unite_ps_ecolos \rightarrow g_credible$
- (G3) $g_credible \rightarrow unite_ps_ecolos$
- (G4) $(crise \wedge g_credible) \rightarrow d_perd_2012$

Etape 2. Formuler le bon problème logique

Pb1 : $\{G1, G2, G3, G4\}$ sat ?

Pb2 : $\{G1, G2, G3, G4, rigueur, g_gagne_region\}$ a pour conséquence d_perd_2012 ?

Pb3 : Modèles de $\{G1, G2, G3, G4, rigueur, g_gagne_region, (NON\ d_perd_2012)\}$ (= valuations qui satisfont l'ensemble)

Etape 3. Résoudre ce problème, « à la main » ou grâce à un « démonstrateur »

2. Codage en Haskell

```
data Form = VAR String | ET Form Form | OU Form Form |  
           IMP Form Form | NON Form  
           deriving (Show,Eq)
```

-- Exemples

-- $f_0 = p \wedge (q \vee \neg r)$

-- $f_1 = (r \rightarrow p) \rightarrow (q \vee \neg r)$

```
f0 = ET (VAR "p") (OU (VAR "q") (NON (VAR "r")))
```

```
f1 = IMP (IMP (VAR "r") (VAR "p")) (OU (VAR "q") (NON  
    (VAR "r")))
```

Exemple « 2012 »

```
(g1) g_gagne_region → (rigueur → crise)
(g2) unite_ps_ecolos → g_credible
(g3) (crise ∧ g_credible) → d_perd_2012
```

En Haskell

```
g_gagne_region = VAR "g_gagne_region" :: Form
rigueur = VAR    »rigueur" :: Form
...
g1 = IMP g_gagne_region (IMP rigueur crise)
g2 = IMP unite_ps_ecolos g_credible
g3 = IMP (ET crise g_credible) d_perd_2012
```

```
data Form = VAR String | ET Form Form | OU Form
          Form | IMP Form Form | NON Form
          deriving (Show,Eq)
```

- Accès aux composantes d'une formule et test de « forme »

```
-- Test si une formule est une implication
isIMP (IMP _ _) = True
isIMP _ = False          -- clause « attrape tout

-- Test si une formule est réduite à une variable
  logique.
isVar (VAR _) = True
isVar _ = False

-- Etc..
```

```
data Form = VAR String | ET Form Form | OU Form Form |  
          IMP Form Form | NON Form  
          deriving (Show,Eq)
```

- Alternative : calcul du type :

```
-- filtrage de type case.. of  
typeForm :: Form -> String  
typeForm f = case f of  
    (VAR _) -> "var"  
    (ET _ _) -> "conj"  
    (OU _ _) -> "disj"  
    (IMP _ _) -> "implic"  
    (NON _) -> "neg"
```

```
data Form = VAR String | ET Form Form | OU Form Form | IMP
          Form Form | NON  Form
          deriving (Show,Eq)
```

```
-- Acces aux arguments
arg1,arg2 :: Form -> Form
```

```
arg1 (IMP a _) = a
arg1 (OU a _) = a
arg1 (ET a _) = a
arg1 (NON a) = a
```

```
arg2 (IMP _ b) = b
arg2 (OU _ b) = b
arg2 (ET _ b) = b
```

NB. En fait on n'utilisera pas ces fonctions, mais directement le filtrage dans la définition des fonctions visées

```
data Form = VAR String | ET Form Form | OU Form Form | IMP
          Form Form | NON  Form
          deriving (Show,Eq)
```

- Pretty Print

```
---- Affichage infixe
```

```
showForm :: Form -> String
```

```
showForm (VAR p) = p
```

```
showForm (ET f g) = "(" ++ (showForm f) ++ " & " ++ (showForm
  g) ++ ")"
```

```
showForm (OU f g) = "(" ++ (showForm f) ++ " | " ++ (showForm
  g) ++ ")"
```

```
showForm (IMP f g) = "(" ++ (showForm f) ++ " -> " ++
  (showForm g) ++ ")"
```

```
showForm (NON f) = "~ " ++ (showForm f)
```

```
-- On combine avec une écriture à l'écran
```

```
pprint = putStrLn . showForm
```

```
> f1
```

```
IMP (VAR "p") (IMP (VAR "q") (NON (VAR "r")))
```

```
> showForm f1
```

```
"(p -> (q -> ~ r)) »
```

```
> pprint f1
```

```
(p -> (q -> ~ r))
```

```
data Form = VAR String | ET Form Form | OU Form Form | IMP
          Form Form | NON  Form
          deriving (Show,Eq)
```

- Evaluation

```
type Valuation = [(String,Bool)]
val0 = [("p",True),("q",False),("r",True)]
val1 = [("p",True),("q",False),("r",False)]

-- assoc x l = valeur associée au nom x dans la liste
-- associative l
-- Question : forme + robuste ?

assoc :: (Eq a) => a -> [(a,b)] -> b
assoc x l =
    let [(nom,val)] = filter (\(z,t) -> (z==x)) l in val

> assoc "q" val1
False
```

```
data Form = VAR String | ET Form Form | OU Form Form | IMP
          Form Form | NON  Form
          deriving (Show,Eq)
```

```
val0 = [("p",True),("q",False),("r",True)]
val1 = [("p",True),("q",False),("r",False)]
-- f0 =  $p \wedge (q \vee \neg r)$ 
f0 = ET (VAR "p") (OU (VAR "q") (NON (VAR "r")))
> eval f0 val0
False
> eval f0 val1
True
```

```
-- eval f v = valeur de la formule f pour la valuation v
```

```
eval :: Form -> Valuation -> Bool
```

```
eval (VAR s) v = assoc s v
```

```
eval (ET f g) v = (eval f v) && (eval g v)
```

```
eval (OU f g) v = (eval f v) || (eval g v)
```

```
eval (IMP f g) v = (not (eval f v)) || (eval g v)
```

```
eval (NON f) v = not (eval f v)
```

```
data Form = VAR String | ET Form Form | OU Form Form |
          IMP Form Form | NON Form          deriving (Show,Eq)
```

- Elimination de l'implication

On utilise l'équivalence : $A \rightarrow B = \neg A \vee B$

```
-- f2 : (r ∨ p) → (q → ¬r)
f2 = IMP (OU (VAR "r") (VAR "p"))
        (IMP (VAR "q") (NON (VAR "r"))))
> elimImp f2          -- ¬(r ∨ p) ∨ (¬q ∨ ¬r)
OU (NON (OU (VAR "r") (VAR "p")))
   (OU (NON (VAR "q")) (NON (VAR "r")))
```

```
elimImp :: Form -> Form
```

```
elimImp (VAR s) = VAR s
```

Analyse -> f et g

Construction -> ET ...

```
elimImp (ET f g) = ET (elimImp f) (elimImp g)
```

```
elimImp (OU f g) = OU (elimImp f) (elimImp g)
```

```
elimImp (IMP f g) = OU (NON (elimImp f)) (elimImp g)
```

```
elimImp (NON f) = NON (elimImp f)
```

NB. Le filtrage permet l'analyse de la formule donnée