Chapitre 5 - Logique

Léo SAMUEL

20 janvier 2021

1 Calcul booléen

1.1 Notations

— Vrai : $\top$
— Faux : $\bot$
— Non : $\neg$
— Et : $\wedge$
— Ou : $\vee$

De plus, dans ce chapitre, on notera $\mathcal{B} = \{\top, \bot\}$
L'ordre de priorité dans les calculs sera : $\neg$,$\wedge$,$\vee$

1.2 Règles de calcul

Propriété 1 *$\wedge$:*
— *Commutatif*
— *Associatif*
— *Neutre :* $\top$

Propriété 2 *$\vee$:*
— *Commutatif*
— *Associatif*
— *Neutre :* $\bot$

Propriété 3 *Entre $\vee$, $\wedge$ et $\neg$*
— *Distributivité ($\vee$ est distributif sur $\wedge$ et inversement contrairement aux nombres)*
— *Loi de De Morgan :* $\forall (a,b,c) \in \mathcal{B}^3$
  — $\neg(a \wedge b) = \neg a \vee \neg b$
  — $\neg(a \vee b) = \neg a \wedge \neg b$

Démonstration 1 *Montrons que* $\forall (a,b,c) \in \mathcal{B}^3, a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
On étudie toutes les possibilités pour $((a,b,c))$. *Il y en a* 8 *car* $2^3 = 8$
On les regroupes dans une table de vérité

a	b	c	$b \vee c$	$a \wedge (b \vee c$	$a \wedge b$	$a \wedge c$	$(a \wedge b) \vee (a \wedge c)$
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0	*0*	*1*	*1*	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*0*	*0*	*0*	*0*
0	*1*	*1*	*1*	*0*	*0*	*0*	*0*
1	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*1*	*1*	*1*	*0*	*1*	*1*
1	*1*	*0*	*1*	*1*	*1*	*0*	*1*
1	*1*	*1*	*1*	*1*	*1*	*1*	*1*

Les colonnes sont identiques

Les propriétés de $\wedge$ et $\vee$ ressemblent à celle pour $\cdot$ et $+$ sur Z/nZ

On utilise alors souvent :
— 0
— 1
— $\cdot$
— $+$
Attention dans ce cas, $1 + 1 = 1$
Dans ce cas, les lois de De Morgan sont : þ Loi de De Morgan : $\forall (a, b, c) \in \mathcal{B}^3$
— $\overline{(a \wedge b)} = \overline{a} + \overline{b}$
— $\overline{(a + b)} = \overline{a}\overline{b}$

Remarque 1 *$(Z/nZ, +, \cdot)$ est un corps.*
$(B, \vee, \wedge)$ n'est même pas un anneau car 1 n'a pas d'opposé

Remarque 2 *Notons $\oplus$ le ou exclusif :*
$1 \oplus 1 = 0$ *On peut vérifier que $(B, \oplus, \wedge)$ est un corps , isomorphe à Z/nZ*

1.3 Autres connecteurs logiques

Combien y-a-t-il de fonction de $\mathcal{B}$ dans $\mathcal{B}$?
Il y en a $|\mathcal{B}|^{|\mathcal{B}|} = 4$
Il y a $id, \neg, 0, 1$

Combien y-a-t-il de fonction de $\mathcal{B}^2$ dans $\mathcal{B}$? Il y en a 16. Nous connaisons déja $\vee, \wedge$,
Pour definir un opérateur, il suffit de donnée sa table de vérité

1.3.1 L'implication

Définition 1 *$\forall a, b \in \mathcal{B}$, on note $a \Rightarrow b$ le booléen $b \vee \neg a$*

Exemple 1 *"Mange ta soupe ou va dans ta chambre" ©FP*
donne
"Si tu ne mange pas ta soupe alors va dans ta chambre"

Interpretation en mathématiques :
Table de vérité de $\Rightarrow$:

a	b	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

Ainsi, la formule $\forall (a, b) \in \mathcal{B}^2, a \Rightarrow b$ est vraie si et seulement si a chaque fois que a est vrai, b l'est aussi. (Quand a est faux, b peut valoir n'importe quoi !)

Digressions :
Un théorème est une formule $\forall x \in E, P(x) \Rightarrow Q(x)$ où E est un ensemble et P, Q des prédicats sur E. (fonctions de E dans $\mathcal{B}$) Dans le cas où P est faux, on a aucune informations sur Q.

Pour montrer q'un théorème est faux, il faut montrer $\neg(\forall x \in E, P(x) \Rightarrow Q(x))$ cad $\exists x \in E, \neg(P(x) \Rightarrow Q(x))$ cad $\exists x \in E, \neg Q(x) \wedge P(x)$ Ainsi, cela revient à trouver un $x \in E$ pour lequel l'hypothèse est vraie mais la conclusion est fausse. Un tel x s'appelle un contre-exemple

1.3.2 Equivalents

Définition 2 *$\forall (a, b) \in \mathcal{B}^2$, on note $a <$ le booléen $(a \Rightarrow b) \wedge (b \Rightarrow a)$ C'est en réalité la même chose que $=$ pour les booléens*

1.3.3 Non-et

Définition 3 $\forall (a,b) \in \mathcal{B}^2, \neg(a \wedge b)$

Il est utilisé car il coute seulement 3 transistors.
On peut definir les autres opérations à partir du non-et. (cf Exercice 9)

De même on définit non-ou.

En général dans un langage, il est seulement fournit $\neg, \wedge, \vee$ desquels on peut définir les autres.

2 Formules logiques

2.1 Type Caml

Voici un type pour représenter les formules logiques :

```
type formule =
  | Variable of string
  | Non of formule
  | Et of (formule*formule)
  | Ou of (formule*formule)
  | Constante of bool
    ;;
```

Ainsi l'exemple $\overline{a+b} \cdot c$ par :

```
let exemple 1 = Et(Non(Ou(Variable "a",Variable "b")),Variable "c");;
```

Ce type est similaire à un type arbre.

2.2 Définition Mathématique

On définit le concept de formule abstraite.
Soit Σ un ensemble dont les éléments seront appelés des "variable (booléenes)". On définit alors l'ensemble des formules logiques ainsi :

- $\forall x \in \Sigma, x$ est une formule
- $\forall b \in B, b$ est une formule
- $\forall$ formule $f, \overline{f}$ est une formule
- $\forall$ formule $f_1, f_2, f_1 + f_2$ et $f_1 f_2$ sont des formules

Exemple 2 *Soit* $a, b, c \in B$
$\overline{(a+b)}c$ *est une formule*
$\vee \vee a$ *n'est pas une formule*

2.3 Sémantique

Donnons un sens aux formules abstraites définies ci-dessus.

En pratique, on va définir des fonctions qui utilisent ces formules

Définition 4 *Un contexte (ou distribution de vérité ou valuation) sur* Σ *est une fonction de* Σ *dans* B.

Choisir un contexte sur Σ revient à choisir la valeur de chaque variable.

Définition 5 *Soit φ un contexte sur Σ. Notons $\mathcal{F}$ l'ensemble des formule sur Σ.*

On définit la fonction "Evaluation en φ", qu'on note $E_{v\varphi}$
Ainsi :
$\forall b\ inB, E_{v\varphi}(b) = b$
$\forall x\ in\Sigma, E_{v\varphi}(x) = \varphi(x)$
$\forall f in\mathcal{F}, E_{v\varphi}(\neg f) = \neg E_{v\varphi}(f)$
$\forall f_1, f_2 in\mathcal{F}, E_{v\varphi}(f_1 f_2) = E_{v\varphi}(f_1)E_{v\varphi}(f_2)$
$\forall f_1, f_2 in\mathcal{F}, E_{v\varphi}(f_1 + f_2) = E_{v\varphi}(f_1) + E_{v\varphi}(f_2)$

Pour représenter un contexte en Ocaml, utilisons un dictionnaire. Pour des exemples avec peu de variables, utilisons les listes d'association.

```
type contexte = (string * bool) list;;
```

La fonction d'évaluation devient alors :

```
let rec evaluation contexte formule=
        match formule with
        | Constante b -> b
        | Variable x -> List.assoc x contexte
        | Non f ->not (evaluation contexte f)
        | Ou (f1,f2) -> (evaluation contexte f1) || (evaluation contexte f2)
        | Et (f1,f2) -> (evaluation contexte f1) && (evaluation contexte f2)
;;
```

2.4 Vocabulaire

Notons toujours $\mathcal{F}$ l'ensemble des formules. Notons $\mathcal{C}$ l'ensemble des contextes sur Σ

Définition 6 *Soit $f \in \mathcal{F}$*
- *On dit que f est satisfiable lorsque $\exists \varphi \in \mathcal{C} tq E_{v\varphi}(f) = \top$*
- *On dit que f est tautologique lorsque $\forall \varphi \in \mathcal{C} tq E_{v\varphi}(f) = \top$*
- *Soit $g \in mathcal F$. On dit que f est équivalente à g $(f \equiv g)$ lorsque $E_{v\varphi}(f) = E_{v\varphi}(g)$*

Ecrivons une fonction qui teste si une formule est satisfiable :

Méthode naive : calculons la liste de tous les contextes :

```
let rec listeVariables formule=
        match formule with
        | Constante b -> []
        | Variable x -> [x]
        | Non f -> listeVariables f
        | Ou (f1,f2) -> fusion_stricte (listeVariables f1) (listeVariables f2)
        | Et (f1,f2) -> fusion_stricte (listeVariables f1) (listeVariables f2)
;;

let rec listeContexte listeVariables=
        match listeVariables with
        | [] -> [[]]
        | t::q -> (List.map (fun x -> (t,false)::x) (listeContexte q))
                                                    @
```

```
                                        (List.map (fun x -> (t,true)::x) (listeContexte q))
;;

let estSatisfiable formule=
        let variables = listeVariables formule in
        let contextes = listeContexte variables in

        let rec aux contexte =
                match contexte with
                | [] -> false
                | t::q -> (evaluation t formule) || aux q
        in
        aux contextes
;;

let estTautologique formule=
        let variables = listeVariables formule in
        let contextes = listeContexte variables in

        let rec aux contexte =
                match contexte with
                | [] -> true
                | t::q -> (evaluation t formule) && aux q
        in
        aux contextes
;;

let plusLongue l1 l2=
  let a, b = List.length l1,List.length l2 in
  if a > b then
    l1
  else l2
;;

let sontEquivalentes f g=
  let variablesf = listeVariables f in
  let variablesg = listeVariables g in

  let contextes = listeContexte (plusLongue variablesf variablesg) in

  let rec aux contexte =
    match contexte with
    | [] -> true
    | t::q -> ((evaluation t f)=(evaluation t g)) && (aux q)
  in
  aux contextes
;;
```

2.5 Formes normales

— Un minterme est vrai pour un seul contexte ($a\bar{b}c$ est vrai pour $1,0,1$)
— Un maxterme est faux pour un et un seul contexte ($\overline{a}+\overline{b}+c$ est faux pour $1,1,0$)
— La négation d'un minterme est un maxterme et réciproquement.
— La forme normale disjonctive revient à la table de vérité de la formule

Exemple 3 *Soit* $f = \overline{a \Rightarrow b} \vee c$
On a $f \equiv \overline{b}a + c$

Ce n'est pas encore la forme normale car les deux monomes n'utilisent pas toutes les variables.

$f \equiv a\overline{b}c + a\overline{b}\overline{c} + abc + \overline{a}bc + \overline{a}\overline{b}c$
Somme de cinq mintermes alors c'est une somme normale disjonctive
Chaque monomes indique un contexte où f est vraie, autrement dit une ligne de la table de vérité

- Le nombre de mintermes dans la formes normale disjonctive est le nombre de contextes où f est vraie
- Si on connait la table de vérité, on peutr retrouver la forme normale disjonctive
- Une forme normale disjonctive est unique
- Mettre deux formules f et g sous forme normale disjonctive est une méthode infaillble pour savoir si $f \equiv g$
- Idem pour voir si f est une tautologie ou satisfiable ...
- La négation d'une forme normale disjonctive est un forme normale conjonctive
- On peut alors partir d'une forme normale disjonctive de f pour obtenir une forme normale conjonctive de $\overline{f}$
- On part de la forme normale disjonctive de $\overline{f}$ pour avoir une forme normale conjonctif de f
- Chaque maxterme s'appelle une clause
- Le nombre de maxterme est le nombre de contexte ou f est fausse ($\bot$)

3 Problème NP-complets

Ces fonctions permettent de résoudre de nombreux problemes
Mais leurs complexités est supérieurs à 2^n

On n'a jamais découvert d'algorithme de complexité polynomial. Ce problème s'appelle "SAT"
Deux problèmes sont équivalent si l'existence d'un algo de complexité polynomial pour l'un entraine l'existence d'un algo de complexité pour l'autre.

Tout problème équivalent à SAT est appelé "NP-Complet" (cf exercice 12)