Logique

Chapitre 3

1 Fermeture inductive et expressions arithmétiques . . . 1
1.1 Definition . . . 1
1.2 Représentation par un arbre . . . 2
2 Syntaxe des formules logiques . . . 3
2.1 Définition . . . 3
2.2 Représentation par un arbre . . . 4
2.3 Implementation en CAML . . . 5
3 Sémantique des formules logiques . . . 5
3.1 Evaluation . . . 5
3.2 Satisfiabilité . . . 7
3.3 Equivalence logique . . . 8
3.4 Tautologie . . . 9
4 Complément : Formes normales conjonctives et problème SAT . . . 10
4.1 Littéraux, conjonctions et disjonctions . . . 10
4.2 Problème SAT . . . 10

1 Fermeture inductive et expressions arithmétiques

1.1 Definition

On se donne :

— un ensemble E,

— une partie B de E (la base)

— des constructeurs qui sont des fonctions f définie d'une partie de E^k dans E (ont dit qu'ils sont d'arité k).

On veut construire la partie de E de tout ce que l'on obtient en utilisant les éléments de la base et en appliquant les constructeurs (éventuellement plusieurs fois).

Définition 1.1.1 (Stabilité)

Soit Ω un ensemble de constructeurs et X une partie de E, on dit que X est stable par Ω si pour tout constructeur f d'arité k de Ω et tout $(x_1, \ldots, x_k) \in X^k$, $f(x_1, \ldots, x_k) \in X$ (s'il existe)

On peut alors définir la fermeture inductive.

Définition 1.1.2 (Fermeture inductive)

Avec les notations précédentes, on appelle fermeture inductive de B par Ω où Ω est un ensemble de constructeurs, le plus petit ensemble de E stable par Ω contenant B.

Remarque : Il y a deux manières de voir la fermeture inductive :

— Celle des mathématiciens (dite aussi descendante) : la fermeture inductive est l'intersection de tous les ensembles contenant B stables par Ω (il y a au moins E en entier)

— Celle des informaticiens (dite aussi ascendante) : On note $B_0 = B$. On construit alors $\Omega(B_0)$ l'ensemble des images par un élément de Ω des éléments de B_0 puis on pose $B_1 = B_0 \cup \Omega(B_0)$. On procède ainsi de suite : $B_{i+1} = B_i \cup \Omega(B_i)$. Pour finir la fermeture inductive est $\bigcup_{i \in \mathbf{N}} B_i$.

Exemple : Les **expressions arithmétiques** se construisent de la sorte. On prend :

— L'ensemble E est l'ensemble de tous les mots s'écrivant avec des caractères ASCII (pour ne pas s'embêter).

— La base $B = \{a, b, c\}$ est l'ensemble des trois lettres qui apparaissent dans nos expressions.

— Les constructeurs sont alors $+$, $\times$, $-$ et $/$ qui sont tous les quatre d'arité 2.

L'ensemble des expressions arithmétiques est alors la fermeture inductive.

Par exemple

$$(a + b)/(a - c)$$

est obtenue en prenant la division des expressions $(a + b)$ et $(a - b)$.

Exercice : Dans ce cas, que vaut B_1 et B_2 ?

1.2 Représentation par un arbre

Les éléments d'une fermeture inductive se représentent naturellement par un arbre. Un élément est en effet de deux sortes

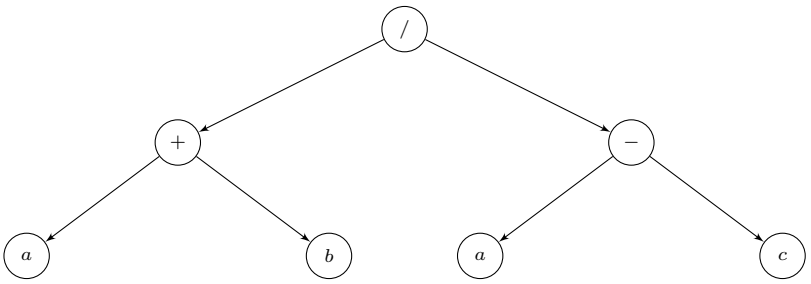
—

—

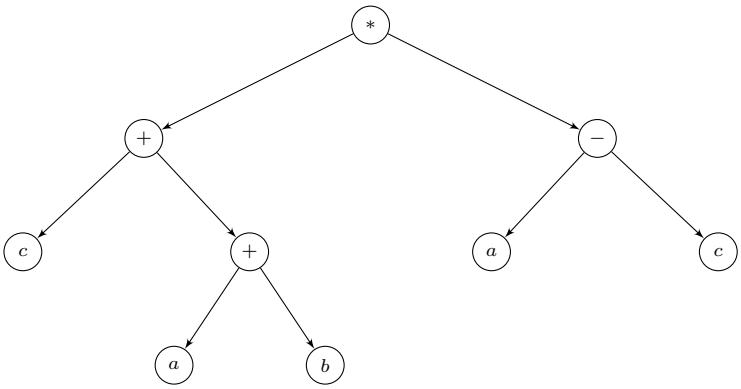
L'expression

$$(a + b)/(a - c)$$

se représente par :



Exercice : Quelle est l'expression représentée par l'arbre suivant ?



Exercice : Donner les arbres correspondant aux expressions : $((a+b)+c)\times c$ et $(a+(b+c))\times c$

Remarque : L'écriture classique des opérations algébriques correspond à la lecture **infixe** de l'arbre (qui est possible car les opérateurs arithmétiques sont d'arité 2 et donc l'arbre est binaire) .

On peut aussi utiliser une écriture **préfixe** des opérateurs.

Dans ce cas l'expression

$$(a+b)/(a-c)$$

s'écrit

On peut aussi utiliser l'écriture **postfixe** qui s'appelle écriture de **Lukasiewicz** ou notation polonaise inverse. L'intérêt est qu'elle ne nécessite pas de parenthèses.

Notre expression s'écrit alors :

Le calcul peut aisément être effectué à l'aide d'une pile.

ATTENTION

Ne pas confondre la syntaxe (abstraite) et la valeur de l'expression. Par exemple $(a+b)+c$ et $a+(b+c)$ sont deux expressions différentes (d'un point de vue de la syntaxe) par contre elles ont la même valeur par associativité de l'addition. Il est est de même pour $a\times(b+c)$ et $a\times b+a\times c$.

2 Syntaxe des formules logiques

2.1 Définition

Les formules logiques vont s'obtenir aussi comme une fermeture inductive.

Définition 2.1.3 (Formule logique)

Soit $\mathscr{V}$ un ensemble (la plupart du temps fini) des variables, les formules logiques forment la fermeture inductive de $B = \mathscr{V} \cup \{V, F\}$ par l'ensemble formé des constructeurs $\wedge, \vee$ d'arité 2 et de $\neg$ d'arité 1.
De ce fait une formule logique est de la forme suivante :

—
—
—
—
—

Terminologie :

1. Les éléments V et F s'appellent les constantes.
2. Les éléments de $\mathscr{V}$ s'appellent les variables propositionnelles.
3. L'opérateur $\wedge$ se lit / se note **et**
4. L'opérateur $\vee$ se lit / se note **ou**.
5. L'opérateur $\neg$ se lit / se note **non**.

Remarques :

1. Il faut faire attention que dans un premier temps on ne donne pas de signification logique à ces expressions. On considère les formules pour elles mêmes. C'est ce que l'on appelle l'aspect *syntaxique*.
2. On s'intéresse dans ce cours à ce que l'on appelle la logique propositionnelle. En particulier nous n'utiliserons de pas quantificateurs qui relèvent de la logique du premier ordre.
3. Il existe des variantes. On peut par exemple utiliser aussi les connecteurs $\Rightarrow$ et $\Leftrightarrow$ tous les deux d'arité de 2.

Exemples :

1. Si on ne considère aucune variables propositionnelles ($\mathscr{V} = \emptyset$), on peut écrire $F, (F \wedge F), \neg(F \vee (F \wedge \neg V))$.
2. Si on considère un ensemble $\mathscr{V} = \{v_1, v_2, v_3\}$ on peut écrire $v_2, (v_2 \wedge \neg v_1)$ ou $((v_1 \vee F) \vee (v_3 \wedge \neg v_3))$.

ATTENTION

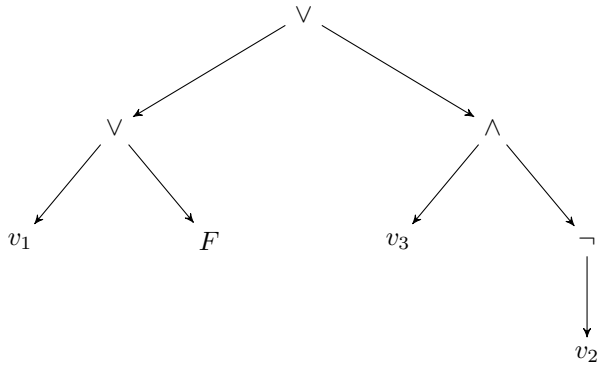
On rappelle que l'on ne donne pas dans un premier de sens aux formules. Par exemples les formules

ne sont pas les mêmes formules

Remarque : On peut éviter de parenthèser les expressions en utilisant les règles de priorités suivantes : $\neg$ est prioritaire sur $\wedge$ et $\wedge$ est prioritaire sur $\vee$. Cependant, dans un premier temps il est plus aisé de garder certaines parenthèses.

2.2 Représentation par un arbre

On peut bien évidemment représenter les formules logiques par des arbres **qui ne seront plus binaires** car l'opérateur $\neg$ est d'arité 1. Par exemple $((v_1 \vee F) \vee (v_3 \wedge \neg v_2))$ se représente par l'arbre :



Remarques :

1. On remarque que les feuilles sont étiquetées avec des variables propositionnelles ou des constantes alors que les nuds sont étiquetés par des constructeurs.
2. L'ecriture classique des formules logiques ne correspondent plus à un parcours usuel de l'arbre. En effet les constructeurs $\wedge$ et $\vee$ s'utilisent de manière **infixe** alors que $\neg$ s'utilise de manière **préfixe.**
3. On peut noter les formules logiques en utilisant la notation **postixe** de l'arbre. Cela a l'avantage de ne plus nécessiter de parenthèses mais cela rend la formule plus compliquée à lire. La formule ci-dessus serait notée

Terminologie :

1. Soit p une formule logique, on appelle sous-formule de p toute formule dont l'arbre correspond à un sous-arbre de celui de p. Par exemple $(v_3 \wedge (\neg v_2))$ est une sous-formule de $((v_1 \vee F) \vee (v_3 \wedge \neg v_2))$.
2. On appelle hauteur d'une formule la hauteur de son arbre associé. Par exemple $((v_1 \vee F) \wedge (v_3 \vee \neg v_2))$ est de hauteur 3. Si on se rappelle de la construction ascendante des fermetures inductives, dire qu'une formule est de hauteur k revient à dire qu'elle appartient à $B_k \setminus B_{k-1}$.

2.3 Implementation en CAML

La definition des formules logiques comme fermeture inductive permet de les implementer simplement un CAML via un type récursif.

```
type formule =
  | V
  | F
  | Var of int
  | Non of formule
  | Ou of formule * formule
  | Et of formule * formule;;
```

3 Sémantique des formules logiques

Maintenant que nous avons construit des formules logiques nous voulons leur « donner un sens ». Précisément on veut savoir si elle est vraie ou fausse. C'est-à-dire lui associer un booléen.
Notation : Par la suite nous noterons $\mathscr{B}$ l'ensemble des booléens. Nous noterons `true` ou 1 pour dire vrai et `false` ou 0 pour dire faux.

3.1 Evaluation

Afin de savoir si une formule logique est vraie ou fausse il faut déjà associer des valeurs booléennes aux variables propositionnelles.

Définition 3.1.4 (Distribution de vérité)

On appelle distribution de vérité une application μ de $\mathscr{V}$ dans l'ensemble $\mathscr{B}$ des booléens. On associe de la sorte une valeur booléenne à chaque variable propositionnelle.

Exemple : Si on reprend $\mathscr{V} = \{v_1, v_2, v_3\}$ on peut par exemple prendre $\mu(v_1) = \mu(v_2) = 1$ et $\mu(v_3) = 0$.
Exercice : Déterminer le nombre de distribution de vérité.

On veut maintenant évaluer une formule logique, ce qui revient à associer un booléen à notre formule. On se base encore sur la structure des formules logiques.

Définition 3.1.5

Soit μ une distribution de vérité, on construit une fonction d'évaluation $\mathscr{E}_\mu$ (ou juste $\mathscr{E}$) qui va de l'ensemble des formules logique dans $\mathscr{B}$. Elle est définie (par induction) par :

- $\mathscr{E}(V) = 1$ *et* $\mathscr{E}(F) = 0$.
- *Si v est une variable propositionnelle,* $\mathscr{E}(v) =$
- *Si p_1 et p_2 sont des formules* $\mathscr{E}(p_1 \wedge p_2) =$
- *Si p_1 et p_2 sont des formules* $\mathscr{E}(p_1 \vee p_2) =$
- *Si p est une formule* $\mathscr{E}(\neg p) =$

Exemple : Reprenons la formule $p = (v_1 \vee F) \wedge (v_3 \vee \neg v_2)$ et la distribution μ ci-dessus. On a donc $\mathscr{E}(v_1 \vee F) = 1$ et $\mathscr{E}(v_3 \vee \neg v_2) = 0$ donc $\mathscr{E}(p) = 0$.
Remarque : Les formules ci-dessus correspondent à la signification classique des connecteurs et et ou.
Exercice : On considère μ telle que $\mu(v_1) = \mu(v_3) = 1$ et $\mu(v_2) = 0$.
Déterminer $\mathscr{E}((\neg v_1 \vee v_2) \wedge \neg(v_1 \wedge v_3))$

Exercice : Écrire une fonction `eval : formule -> (bool array) -> bool` telle que l'instruction `eval f mu` renvoie le booléen $\mathscr{E}_\mu(f)$.

3.2 Satisfiabilité

Définition 3.2.6

Soit $\mathscr{V}$ un ensemble et p une formule sur $\mathscr{V}$.

1. *Soit μ une distribution de vérité, on dit que p est satisfaite pour μ ou que μ satisfait p si $\mathscr{E}_\mu(p) = 1$. On note alors $\mu \vDash p$.*
2. *On dit que la formule p est satisfiable s'il existe une distribution μ telle que p soit satisfaite pour μ.*

Remarque : Dans le cas général il est difficile de savoir si une formule est satisfiable. L'algorithme naif consiste en effet à tester toutes les 2^n distributions de vérités (dans le cas où $\mathscr{V}$ est de cardinal n). On voit donc que c'est un problème de complexité exponentielle.

★ **Méthode :** On sera amené à établir la *table de vérité* d'un formule qui est la table qui donne l'évaluation d'une formule p pour toutes les distributions de vérités (on placera ces dernières dans l'ordre lexicographique pour ne pas en oublier).

Reprenons l'exemple de $p = (v_1 \vee F) \vee (v_3 \vee \neg v_2)$.

v_1	v_2	v_3	$v_1 \vee F$	$v_3 \vee \neg v_2$	p
0	0	0	0	1	1
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	1	1

Exercice : Donner les tables de vérités de $\neg v$, $v_1 \wedge v_2$, $v_1 \vee v_2$.

3.3 Equivalence logique

Définition 3.3.7

Soit p_1 et p_2 deux formules logiques sur le même ensemble $\mathscr{V}$. On dit que p_1 et p_2 sont équivalentes et on note $p_1 \equiv p_2$ si pour toute distribution de vérité μ, $\mathscr{E}_\mu(p_1) = \mathscr{E}_\mu(p_2)$.

Remarque : Cela signifie que p_1 est vraie si et seulement si p_2 l'est aussi. C'est-à-dire :

$$\forall \mu \in \mathscr{B}^{\mathscr{V}}, \mu \vDash p_1 \iff \mu \vDash p_2.$$

Exemples :

1. Les formules $v_1 \wedge v_2$ et $v_2 \wedge v_1$ sont équivalentes. Il suffit d'établir les tables de vérités.
2. Les formules v_1 et $v_1 \vee (v_2 \wedge \neg v_2)$ sont équivalentes. Il suffit d'établir les tables de vérités.

ATTENTION

Ne pas confondre des formules égales et des formules équivalentes. Les deux formules

sont équivalentes mais elles ne sont pas égales.

Définition 3.3.8

Soit p_1 et p_2 deux formules, on utilise les abréviations suivantes :

- *on note $p_1 \Rightarrow p_2$ pour $p_2 \vee \neg p_1$*
- *on note $p_1 \Leftrightarrow p_2$ pour $(p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$*
- *on note p_1 xor p_2 pour $(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)$*

Exercice : Déterminer, via des tables de vérités les valeurs de $p_1 \Rightarrow p_2$, $p_1 \Leftrightarrow p_2$ et p_1 xor p_2 en fonction de celles de p_1 et p_2.

Remarque : Quand on ne s'intéresse qu'à la sémantique et pas à la syntaxe, on pourra utiliser ces notations pour des formules qui y sont équivalentes. Par exemple on pourra noter $v_2 \Rightarrow v_1$ pour $v_1 \vee ((\neg v_2 \wedge v_1) \vee (\neg v_2 \wedge \neg v_1))$

3.4 Tautologie

Définition 3.4.9

Une formule logique définie sur $\mathscr{V}$ est une tautologie si elle est satisfaite pour tout distribution de vérité.

Remarques :

1. Une tautologie est donc quelque chose qui est toujours vrai
2. On appelle à l'inverse une antilogie une formule qui n'est pas satisfiable.
3. Si p_1 et p_2 sont des propositions logiques, $p_1 \equiv p_2$ si et seulement si $p_1 \Leftrightarrow p_2$ est une tautologie.

ATTENTION

Ne pas confondre « p est satisfiable » (**il existe** μ telle que $\mu \vDash p$) avec « p est une tautologie » (**pour tout** μ, $\mu \vDash p$).

Proposition 3.4.10 (Propriétés de $\wedge$)

On désigne par p_1, p_2 et p_3 des formules logiques. Les propositions suivantes sont des tautologies

- $(p_1 \wedge F) \iff$
- $(p_1 \wedge V) \iff$
- $(p_1 \wedge p_2) \iff$
- $(p_1 \wedge (p_2 \wedge p_3)) \iff$
- $(p_1 \wedge p_1) \iff$

Proposition 3.4.11 (Propriétés de $\vee$)

On désigne par p_1, p_2 et p_3 des formules logiques. Les propositions suivantes sont des tautologies :

- $(p_1 \vee F) \iff$
- $(p_1 \vee V) \iff$
- $(p_1 \vee p_2) \iff$
- $(p_1 \vee (p_2 \vee p_3)) \iff$
- $(p_1 \vee p_1) \iff$

Proposition 3.4.12 (Autres tautologies)

On désigne par p_1, p_2 et p_3 des formules logiques. Les propositions suivantes sont des tautologies :

- $(p_1 \wedge (p_2 \vee p_3)) \iff$
- $(p_1 \vee (p_2 \wedge p_3)) \iff$
- $\neg(p_1 \wedge p_2) \iff$
- $\neg(p_1 \vee p_2) \iff$
- $(p_1 \vee \neg p_1) \iff$
- $(p_1 \wedge (p_1 \Rightarrow p_2)) \Rightarrow$

4 Complément : Formes normales conjonctives et problème SAT

Comme on l'a vu, si on s'intéresse à la sémantique, une « même » formule logique peut s'écrire de différentes manières. Par exemple $(p_1 \wedge (p_2 \vee \neg p_2))$ est équivalente à p_1. On peut se demander de quelle manière standardisée on peut écrire une formule logique.

4.1 Littéraux, conjonctions et disjonctions

Définition 4.1.13

1. *On appelle littéral une formule logique de la forme v ou $\neg v$ où $v \in \mathscr{V}$.*
2. *On appelle clause un disjonction de littéraux c'est-à-dire une formule logique c telle qu'il existe $\ell_1, \dots, \ell_p$ des littéraux tels que $c = \bigvee_{i=1}^{p} \ell_i$.*
   Pour tout entier n, si la clause contient moins de n littéraux on dit qu'il s'agit d'une n-clause.
3. *On appelle forme normale conjonctive d'une formule logique f, une formule logique qui est une conjonction de clauses et qui est équivalente à p. C'est-à-dire la donnée de clauses $c_1, \dots, c_q$ telles que*

$$f \equiv \bigwedge_{i=1}^{q} c_i$$

Théorème 4.1.14

Toute formule logique admet des formes normales conjonctives.

Démonstration : Voir l'exercice 12.
Remarque : Il faut faire attention que si on applique « simplement » les formules de distributivité et les lois de De Morgan on peut obtenir des formes normales conjonctives dont nombre de clauses croit exponentiellement en fonction de la taille de la formule initiale.

4.2 Problème SAT

Le problème SAT est le problème de décision qui consiste à savoir si une formule logique est satisfiable, plus précisément, étant donné une formule logique f, existe-t-il une distribution de vérité μ telle que $\mu \vDash f$.

De manière générale, il y a deux classes importantes de problèmes de décision :

— La classe P qui consiste en les problèmes de décision qui peuvent être résolus en temps polynomial par rapport à la taille des entrées.

— La classe NP qui consiste en les problèmes de décision pour lesquels on peut vérifier en temps polynomial l'exactitude d'une solution.

On peut vérifier simplement que $P \subset NP$. L'inclusion réciproque est probablement le résultat le plus important en informatique fondamentale. Il fait parti des problèmes du millénaire de l'institut Clay.

On peut voir que le problème SAT est dans la classe NP, quand on a une formule logique et une distribution de vérité, on peut l'évaluer en temps polynomial.

Par contre, l'algorithme naïf pour vérifier qu'une formule est satisfiable consiste à tester toutes les distributions de vérités ce qui est donc exponentiel par rapport au nombre des variables.

De fait, le problème SAT est un problème NP-complet, cela signifie que si on arrive à montrer qu'il appartient à la classe P alors on aura le problème P=NP car tous les problèmes NP peuvent se réduire (en temps polynomial) au problème SAT (théorème de Cook). De plus, le problème 3-SAT qui est le problème SAT pour les formes normales conjonctives de 3-clauses est déjà un problème NP-complet (alors que l'on sait trouver un algorithme polynomial pour résoudre 2-SAT).