Logique propositionnelle #1

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Syntaxe ?

« Que est ce doncques que sintaxe ? »

Syntaxe v. sémantique

Informellement…

Syntaxe

- « forme » d'un objet
- informatiquement, par ex. défini par un type
  - mais peut aussi l'être par des outils plus complexes, par ex. des *grammaires*
- monde *formel*

Sémantique

- « sens » que l'on donne à un objet ; peut faire le lien entre différents types d'objets
- informatiquement, par ex. défini par des fonctions d'évaluation
- monde philosophique

Exemple 1 : encore une fois les entiers naturels à la Peano

- On peut définir le :

  ```
  type nat = O | S of nat
  ```

  hors de tout contexte

- On peut *choisir* d'interpréter `O` comme la notion sensible de l'entier naturel zéro, `S O` comme la notion sensible de l'entier naturel un, etc.

- Faire ce choix, c'est associer une *sémantique* à `nat`

- Ce choix n'est pas unique ; on pourrait par ex. associer la notion sensible de « valeur fausse » à `O`, et de « valeur vraie » à n'importe quel `nat` construit avec `S`

Exemple 1 *bis* : sémantique par évaluation

- Si l'on admet que les `int` et les `bool` représentent la notion sensible d'entier relatif et de valeurs vraie/faux (en fait c'est *aussi* un choix), on peut implémenter les sémantiques ci-dessus :

```
let rec evalint = function O -> 0 | S n -> 1 + (evalint n)

let evalbool = function O -> false | S _ -> true
```

Exemple 2 : TP18 Exercice 1

Go and see

Table des matières

Formules de la logique propositionnelle

Très informellement :

- Les trucs avec des « et », des « ou » etc. mais pas de quantificateurs

Formellement : *via* une **syntaxe**

Formules de la logique propositionnelle

Soit $\mathcal{V}$ un ensemble de variables (ou *atomes*), une *formule de la logique propositionnelle* (ou *proposition*) sur $\mathcal{V}$ est ou bien :

- $\top$ (formule triviale)
- $\bot$ (formule anti-triviale)
- v pour $v \in \mathcal{V}$
- $\neg\varphi$ pour φ une formule
- $\varphi_1 \wedge \varphi_2$; $\varphi_1 \vee \varphi_2$; $\varphi_1 \to \varphi_2$ pour φ_1, φ_2 des formules

ainsi qu'éventuellement : (φ) pour φ une formule, et le sucre syntaxique $\varphi_1 \leftrightarrow \varphi_2$ qui n'est qu'une notation pour $(\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$

Formules de la logique propositionnelle *bis*

Définition inductive

La définition ci-dessus n'est rien d'autre qu'une définition inductive avec :

- $\mathcal{B} = \{\top, \bot, \} \cup \mathcal{V}$
- $I = \{\neg, \wedge, \vee, \rightarrow\}$ où $\neg$ est unaire préfixe, et $\wedge$, $\vee$, $\rightarrow$ binaires infixes
  - La notation infixe est la plus usuelle, mais l'on pourrait par ex. très bien définir une syntaxe alternative avec notation préfixe ou postfixe

Non-ambiguïté

- Pour que la définition soit non-ambiguë, on parenthèse les arguments des constructeurs : par ex., la formule construite avec $\rightarrow$ appliqué à φ_1 et φ_2 est $(\varphi_1) \rightarrow (\varphi_2)$
- En pratique on pourra *visuellement* omettre certaines parenthèses en définissant des priorités sur les constructeurs : $\neg$ prioritaire sur $\wedge$ prioritaire sur $\vee$ prioritaire sur $\rightarrow$:
  - $a \vee b \wedge c$ est un raccourci visuel pour $(a) \vee ((b) \wedge (c))$
  - $a \vee b \rightarrow c$ est un raccourci visuel pour $((a) \vee (b)) \rightarrow (c)$
- Mais à l'exception des parenthèses autour des cas de base et $\neg$, mieux vaut maintenir la plupart : l'objectif premier est d'être clair

Formules de la logique propositionnelle *ter*

Représentation OCaml

La définition inductive peut se représenter (syntaxiquement) par un type récursif :

```
type lf =
    | Top
    | Bot
    | V of int
    | Neg of lf
    | And of lf * lf
    | Or  of lf * lf
    | Imp of lf * lf
```

où l'on a ici identifié $\mathcal{V}$ aux valeurs de type `int`

Remarques

- Une formule est alors représentée par un arbre (non-ambigu)
- On aurait pu commencer par définir les formules comme des arbres, et dériver une définition « à plat » plutôt que l'inverse

Exercice

Soit le type d'arbre unaire/binaire strict :

```
type ('a, 'b, 'c) bt =
    | L of 'a
    | N1 of 'b * ('a, 'b, 'c) bt
    | N2 of 'c * ('a, 'b, 'c) bt * ('a, 'b, 'c) bt
```

- Définissez trois types `ato`, `co1`, `co2` tels que `lf` soit *isomorphe* à `(ato, co1, co2) bt`
- Écrivez les fonctions `lf2bt : lf -> (ato, co1, co2) bt` et `bt2lf : (ato, co1, co2) bt -> lf` implémentant cet isomorphisme

Exercice : correction

Par exemple :

```
type ato = To | Bo | Va of int
type co1 = Ne
type co2 = An | O | Im

let rec lf2bt = function
    | Top -> L To
    | Bot -> L Bo
    | V x -> L (Va x)
    | Neg x -> N1 (Ne, lf2bt x)
    | And (x, y) -> N2 (An, (lf2bt x), (lf2bt y))
    | Imp (x, y) -> N2 (Im,  (lf2bt x), (lf2bt y))
    | Or  (x, y) -> N2 (O,   (lf2bt x), (lf2bt y))
```

`bt2lf` similaire

Aplatissement

- Si l'on souhaite aplatir une formule (représentée par un arbre), par ex. pour l'afficher « joliment », il suffit d'effectuer un parcours :

```
let rec pp = function
    | Top -> "T" | Bot -> "_"
    | V i -> "v"(string_of_int i)
    | Neg f -> "~("(pp f)")"
    | And (x, y) -> "("(pp x)") /\\ ("(pp y)")"
    | Or  (x, y) -> "("(pp x)") \\/ ("(pp y)")"
    | Imp (x, y) -> "("(pp x)") -> ("(pp y)")"
```

Vocabulaire

Variables, connecteurs

- Les éléments de $\mathcal{V}$ sont des *variables propositionnelles*
- Les constructeurs $\neg$, $\wedge$, $\vee$, $\rightarrow$ sont aussi appelés *connecteurs logiques*

Sous-formule

- Les sous-arbres d'une représentation arborescente d'une formule propositionnelle en sont des *sous-formules*
- De façon équivalente, soit une formule φ définie inductivement de façon non-ambiguë, tout $\varphi' < \varphi$ pour l'ordre induit en est une sous-formule

Hauteur

La *hauteur* d'une formule peut se définir par exemple :

- comme la hauteur de son arbre dans une représentation arborescente (comme pour les arbres, plusieurs conventions sont possibles)
- comme sa hauteur en tant qu'élément d'un ensemble défini inductivement (ce qui est essentiellement la même chose)

Égalité syntaxique

Égalité syntaxique

En tant qu'objets syntaxiques, deux formules sont *égales* (tout simplement) si elles désignent le même objet

- Par exemple, dans une représentation par un type arborescent, si les valeurs sont les mêmes (arbres)

Exemples

- $a \vee b$ est égale à $a \vee b$
- $a \vee b$ n'est pas égale à $b \vee a$
- $a \wedge \neg a$ n'est pas égale à $\bot$
- $a \vee \neg a$ n'est (surtout pas) égale à $\top$

Égalité modulo

On peut éventuellement définir une égalité syntaxique modulo (par ex.) commutativité pour les connecteurs $\vee$ et $\wedge$, mais on s'en abstiendra généralement

Représentation par des types 

On peut représenter les formules propositionnelles par des types OCaml :

- $\top$ représenté par `type top = Top`
- $\bot$ représenté par `type bot = |`
- $\mathcal{V}$ identifié aux variables de types polymorphes

Puis pour `t1`, `t2` les types représentant φ_1 et φ_2 :

- $\varphi_1 \wedge \varphi_2$ représenté par `t1 * t2`
- $\varphi_1 \vee \varphi_2$ représenté par `(t1, t2) Either.t`
- $\varphi_1 \to \varphi_2$ représenté par `t1 -> t2`
- $\neg\varphi_1$ représenté par `t1 -> bot`

Cette représentation a d'énormes avantages, mais n'est pas au programme :(

Court bilan sur la syntaxe

- On a défini (plusieurs variantes) des formules propositionnelles, *en tant qu'objets formels*
- On n'a pour l'instant associé aucun *sens* particulier à ces objets
- Vous verrez l'année prochaine des règles purement syntaxiques pour manipuler ces formules, associées à des notions de *théorèmes* et de *démonstrations*
  - Pour la représentation par des types (malheureusement hors programme), la démonstration d'une formule correspond à un programme dans un certain langage fonctionnel qui en possède le type

Table des matières

Objectif

- On veut utiliser les formules propositionnelles pour modéliser des « liens logiques » entre propositions (plus petites)
- Une sémantique pour la logique propositionnelle doit préciser le sens donné aux symboles (notamment les connecteurs) utilisés
  - Quel lien logique cherche-t on à exprimer par $v_0 \wedge v_1$? Par $v_0 \to v_1$?
- Une sémantique *n'a pas* à donner de sens à ce que les variables modélisent
  - v_0 veut-elle dire « il fait beau » ou « tout corps fini est commutatif » ?
- *Il existe plusieurs sémantiques pour la logique propositionnelle, qui ne sont pas équivalentes*

Sémantique par tableau de vérité

- L'unique sémantique au programme est celle par table(au) de vérité (booléen)
- Modélise les *règles de raisonnement* **classiques**
  - *Cf.* cours de l'année prochaine sur la *déduction naturelle*
  - (« classique » : admet le *principe du tiers-exclu* ou une règle de raisonnement équivalente, en opposition par ex. à la logique *intuitionniste*, qui le refuse)

Principe

On définit inductivement si une formule est *vraie* ou *fausse* en fonction des valeurs de vérité de ses sous-formules

- Les cas de base correspondent aux assertions (sans sous-formules : *éléments minimaux* pour l'ordre induit)
  - $\top$: toujours vraie
  - $\bot$: toujours fausse
  - $v_i \in \mathcal{V}$: véracité définie par une *fonction de valuation*
- Les autres constructeurs spécifient *via* leur *table* (ou *tableau*) de vérité à quelles conditions sur leurs sous-formules il construisent une formule vraie

Sémantique par t.v. *bis*

Valuation des variables

Une *valuation* (ou *affectation*) est une application $\sigma : \mathcal{V} \to \{V, F\}$ des variables vers les « valeurs de vérité » « vrai » et « faux » (notées V & F, ou 1 & 0, ou `true` & `false`, ou …)

- Le programme officiel dit V pour « vrai », F pour « faux », mais 1 et 0 sont nettement plus pratiques et l'on se permettra souvent de les utiliser à la place
- Pour n variables, il y a 2^n valuations différentes de l'ensemble des variables ; on notera $\mathbb{V}$ l'ensemble de ces valuations

Représentation informatique d'une valuation

Typiquement par extension :

- Si les variables sont identifiées par des entiers naturels consécutifs (généralement le cas), un tableau de `bool` ou `int` suffit
  - Voire par exemple un unique `uint64_t` si l'on a ≤ 64 variables
- Sinon, un tableau associatif peut être nécessaire / suffire

Évaluation d'une formule (début)

On peut déterminer les valeurs de vérité des cas de base, en fonction d'une valuation donnée en argument :

```
let rec eval v
    = function
    | Top -> true
    | Bot -> false
    | V i -> v.(i)
    | _ -> failwith "not implemented"
```

Sémantique par t.v. *ter*

Représentation informatique des tables de vérité

On peut simplement définir des `function`s qui traitent toutes les valeurs en entrée possibles

Évaluation d'une formule (fin)

Dans les cas inductifs, il suffit d'appliquer les tables de vérité associées données en argument ; c'est en fait un *fold* :

```
let rec eval tvneg tvand tvor tvimp v f =
    let eval' = eval tvneg tvand tvor tvimp v in match f with
    | Top -> true
    | Bot -> false
    | V i -> v.(i)
    | Neg f -> tvneg (eval' f)
    | And (f1, f2) -> tvand (eval' f1, eval' f2)
    | Or  (f1, f2) -> tvor  (eval' f1, eval' f2)
    | Imp (f1, f2) -> tvimp (eval' f1, eval' f2)
```

Sémantique par t.v. *quater*

Tableaux de vérité des connecteurs

En logique classique, on choisit que :

- $\neg\varphi$ est vraie ssi. φ est fausse
- $\varphi_1 \wedge \varphi_2$ est vraie ssi. φ_1 et φ_2 sont toutes les deux vraies
- $\varphi_1 \vee \varphi_2$ est vraie ssi. au moins l'une de φ_1 et φ_2 est vraie
- $\varphi_1 \rightarrow \varphi_2$ est vraie sauf si φ_1 est vraie et φ_2 est fausse

Implémentation

```
let cneg = function true -> false | false -> true
let cand = function true, true   -> true  | _ -> false
let cor  = function false, false -> false | _ -> true
let cimp = function true, false  -> false | _ -> true
let evalc = eval cneg cand cor cimp
```

Tableau de vérité d'une formule

Le tableau de vérité d'une formule f sur $\mathcal{V}$ énumère `evalc v f` pour chacune des $2^{\#\mathcal{V}}$ valuations v

Notation ; exemples

Notation

On pourra noter $[\![\varphi]\!]_\sigma$ l'évaluation de φ dans la sémantique des tableaux de vérité booléens pour l'affectation σ *(notation pas au programme)*

Exemples

Sémantique par t.v. : présentation alternative

Valuation des variables

- $\mathcal{V}$ ensemble des variables, $\mathbb{V}$ ensemble des valuations v
- $v : \mathcal{V} \to \{0, 1\}$: valuation 1 (V) ou 0 (F) de chaque variable

Évaluation d'une formule

v induit $\hat{v}$ sur les formules par induction comme :

- $\hat{v}(\bot) = 0$
- $\hat{v}(\top) = 1$
- $\hat{v}(x) = v(x)$ pour toute variable $x \in \mathcal{V}$
- $\hat{v}(\neg A) = 1 - \hat{v}(A)$
- $\hat{v}(A \wedge B) = \min(\hat{v}(A), \hat{v}(B))$
- $\hat{v}(A \vee B) = \max(\hat{v}(A), \hat{v}(B))$
- $\hat{v}(A \to B) = \max(1 - \hat{v}(A), \hat{v}(B))$

Modèles

Modèle

Un *modèle* d'une formule φ pour la sémantique par tableau de vérité est une valuation σ des variables telle que $[\![\varphi]\!]_\sigma = V$ (une valuation pour laquelle la table de vérité de φ « donne V »)

Notation : Pour σ une valuation, φ une formule, on note :

- $\sigma \vDash \varphi$ pour « σ est un modèle de φ »
- $\sigma \nvDash \varphi$ pour « σ n'est pas un modèle de φ »
- $\mathbb{M}(\varphi)$ pour l'ensemble des modèles de φ
  - $\mathbb{M}(\varphi) = \{\sigma \mid \sigma \vDash \varphi\}$

(Dans un contexte où l'on peut considérer plusieurs sémantiques, il faudrait enrichir la notation pour éviter toute ambiguïté, par ex. écrire « $\sigma \vDash_{\text{tv}} \varphi$ »)

Extension aux ensembles de formules

Soit Γ un ensemble de formules, les modèles de Γ sont donnés par l'intersection des modèles des formules : $\mathbb{M}(\Gamma) = \bigcap_{\varphi \in \Gamma} \mathbb{M}(\varphi)$

Quelques propriétés des modèles

De *evalc* et des tables de vérité des connecteurs, on déduit que l'on a pour toutes formules φ, φ_1 et φ_2 :

- $\mathbb{M}(\bot) = \varnothing$
- $\mathbb{M}(\top) = \mathbb{V}$
- $\mathbb{M}(\neg\varphi) = \mathbb{V} \setminus \mathbb{M}(\varphi)$
- $\mathbb{M}(\varphi_1 \wedge \varphi_2) = \mathbb{M}(\varphi_1) \cap \mathbb{M}(\varphi_2)$
- $\mathbb{M}(\varphi_1 \vee \varphi_2) = \mathbb{M}(\varphi_1) \cup \mathbb{M}(\varphi_2)$
- $\mathbb{M}(\varphi_1 \to \varphi_2) = \mathbb{M}(\neg\varphi_1 \vee \varphi_2) = (\mathbb{V} \setminus \mathbb{M}(\varphi_1)) \cup \mathbb{M}(\varphi_2)$

(admis)

SAT

Satisfiabilité

Une formule φ est :

- satisfiable (SAT) si elle admet (au moins) un modèle ($\mathbb{M}(\varphi) \neq \varnothing$)
- insatisfiable (UNSAT) si elle n'admet aucun modèle ($\mathbb{M}(\varphi) = \varnothing$)

Problème SAT : « étant donnée φ, décider si elle est satisfiable »

- Algorithme évident : tester les 2^n valuations possibles ; on sait faire mieux (*cf.* TP), mais c'est un problème difficile

Tautologie

Une formule φ est :

- une *tautologie* si toutes ses valuations sont des modèles ($\mathbb{M}(\varphi) = \mathbb{V}$)
  - notation : $\vDash \varphi$
- une *antilogie* si elle est la négation d'une tautologie (si elle est insatisfiable)
  - notation : $\nvDash \varphi$

$\top$ (resp. $\bot$) est une tautologie (resp. antilogie) triviale

(D'un point de vue formel, les tautologies correspondent aux *théorèmes* de la déduction naturelle pour la logique **classique**, *cf.* l'année prochaine)

Équivalence

Équivalence sémantique

Deux formules φ_1 et φ_2 sont *sémantiquement* (ou *logiquement*) *équivalentes*, noté $\varphi_1 \equiv \varphi_2$ ssi. elles admettent les mêmes modèles : $\mathbb{M}(\varphi_1) = \mathbb{M}(\varphi_2)$

Remarque importante

Deux formules équivalentes ne sont pas nécessairement syntaxiquement égales

Exemples

- Toutes les tautologies (resp. antilogies) sont sémantiquement équivalentes entre elles
- $v_0 \wedge v_1 \equiv v_1 \wedge v_0$
- $v_0 \rightarrow v_1 \equiv (\neg v_0) \vee v_1$

Réductions

- Réduction de SAT à l'équivalence :
  1. φ UNSAT ssi. $\varphi \equiv \bot$
- Réduction de l'équivalence à SAT :
  1. $\varphi_1 \equiv \varphi_2$ ssi. $\vDash \varphi_1 \leftrightarrow \varphi_2$ (admis)
  2. $\vDash \varphi$ ssi. $\nvDash \neg\varphi$ (c'est à dire $\neg\varphi$ UNSAT)

Équivalence & SAT : algorithme évident en OCaml

```
let int2array n x =
    Array.init n
    (fun i -> ((x lsr i) land 1) = 1)

let equiv n f1 f2 =
    let rec all_same i =
        if i = 1 lsl n then true
        else let ev = evalc (int2array n i) in
             (ev f1 = ev f2) && all_same (succ i)
    in
    all_same 0

let sat n f = not (equiv n f Bot)
```

Quelques équivalences usuelles

Pour toutes formules φ, φ_1, φ_2, φ_3 on peut prouver (par exemple par calcul, pour toutes les valeurs de vérité) que l'on a :

Neutralité

- $\varphi \wedge \top \equiv \varphi$
- $\varphi \vee \bot \equiv \varphi$

Idempotence

- $\varphi \wedge \varphi \equiv \varphi$
- $\varphi \vee \varphi \equiv \varphi$

Distributivité

- $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
- $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

Quelques équivalences ou tautologies *classiques*

De même :

Cohérence

- $\vDash \neg(\varphi \wedge \neg\varphi)$

Tiers-exclu

- $\vDash \varphi \vee \neg\varphi$

Élimination de la double négation

- $\neg\neg\varphi \equiv \varphi$

Lois « de De Morgan »

- $\neg(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \wedge \neg\varphi_2$
- $\neg(\varphi_1 \wedge \varphi_2) \equiv \neg\varphi_1 \vee \neg\varphi_2$

Contraposition

- $\varphi_1 \to \varphi_2 \equiv \neg\varphi_2 \to \neg\varphi_1$

Conséquence

Conséquence sémantique

Une formule φ_2 est *conséquence sémantique* (ou *logique*) d'une formule φ_1 si les modèles de φ_1 sont inclus dans ceux de φ_2 : $\mathbb{M}(\varphi_1) \subseteq \mathbb{M}(\varphi_2)$. Notation : $\varphi_1 \models \varphi_2$
Autrement dit, si $\sigma \models \varphi_1$, alors $\sigma \models \varphi_2$
Extension aux ensembles de formules : $\Gamma \models \varphi$ ssi. $\mathbb{M}(\Gamma) \subseteq \mathbb{M}(\varphi)$

Exemples

Pour toutes formules φ, φ_1, φ_2, et ensemble de formule Γ :

- $\bot \models \varphi$
- $\varphi \models \top$
- $\varphi \models \varphi$
- $\Gamma \cup \{\varphi\} \models \varphi$
- $\varphi_1 \wedge \varphi_2 \models \varphi_1$
- $\varphi_1 \models \varphi_1 \vee \varphi_2$
- $\varphi_1 \models \varphi_2$ ssi. $\models \varphi_1 \rightarrow \varphi_2$