Logique propositionnelle

Tahina Ramananandro
ramanana@clipper.ens.fr

Lycée Louis-le-Grand – MPSI CAML
21 mai 2008

1 Syntaxe : qu'est-ce qu'une formule ? Comment la représenter ?

Lire attentivement toute cette section et taper « en direct live » le code CAML proposé.

1.1 Définition formelle

Fixons-nous un ensemble V de variables. Alors, une *formule* de la logique propositionnelle sur l'ensemble des variables V peut être :

- soit la constante notée $\top$ et lue « Vrai »
- soit la constante notée$\bot$ et lue « Faux »
- soit une variable propositionnelle, élément de V
- soit la négation d'une formule P, notée $\neg P$, lue « non P »
- soit la conjonction de deux formules P, Q, notée $P \wedge Q$, lue « P et Q »
- soit la disjonction de deux formules P, Q, notée $P \vee Q$, lue « P ou Q »
- soit l'implication de deux formules P, Q, notée $P \Rightarrow Q$, lue « P implique Q »

Exemples de formule (avec $V = \{x, y\}$) : $(x \wedge \top) \Rightarrow \neg (\bot \vee y)$, lue « ($x$ et Vrai) implique non (Faux ou y) »

1.2 Représentation en CAML

1.2.1 Définition du type des formules par un type construit

En CAML, on décrit le type des formules par un *type construit*. En supposant désormais que les variables sont des entiers naturels ($V = \mathbb{N}$), on déclarera alors le type construit des formules :

```
type formule =
| Vrai                            (* constante Vrai *)
| Faux                            (* constante Faux *)
| Variable of int                 (* variable propositionnelle, ici un entier *)
| Non      of formule             (* négation d'une formule *)
| Et       of (formule * formule) (* conjonction de deux formules *)
| Ou       of (formule * formule) (* disjonction de deux formules *)
| Implique of (formule * formule) (* implication de deux formules *)
;;
```

Lorsqu'on définit ce type construit, on définit automatiquement des mots-clés, appelés *constructeurs*, qui permettront en effet de construire des objets ayant ce type. Ici, ces constructeurs seront donc **Vrai**, **Faux**, **Variable**, **Non**, **Et**, **Ou**, **Implique**.

Selon que dans la définition de leur type construit ces constructeurs soient ou non suivis d'une clause **of**, ils attendent un ou zéro argument.

Les constructeurs **Vrai** et **Faux** n'étaient pas suivis de la clause **of** : ils n'attendent donc pas d'argument et peuvent être utilisés seuls. Ainsi, **Vrai** et **Faux** sont des objets de type `formule`.

En revanche, **Variable**, **Non**, **Et**, **Ou**, **Implique** attendent tous un argument. Dans le cas de **Variable**, cet argument doit être un entier : ainsi, `Variable 2` est un objet de type `formule`. Toutefois, **Non** attend un argument de type `formule` : il faut donc déjà disposer d'un tel objet pour pouvoir le passer comme argument à **Non** et obtenir ainsi un nouvel objet de type `formule`. De même, **Et**, **Ou**, **Implique** requièrent un argument qui doit être un couple de formules, et, une fois un tel argument passé, on obtient un nouvel objet de type `formule`.

Une formule est ainsi représentée par un terme du type `formule`. Par exemple, la formule :

$$(2 \wedge \top) \Rightarrow \neg (\bot \vee 3)$$

se représentera donc sous la forme suivante :

```
Implique (Et (Variable 2, Vrai), Non (Ou (Faux, Variable 3)))
```

C'est une représentation analogue à celle d'un *arbre*[1]. (Me rappeler de dessiner l'arbre au tableau). Cet objet est appelé *arbre syntaxique*[2] de la formule.

1.2.2 Utilisation des objets du type construit *formule*

La fonction suivante prend en argument une formule (représentée par un terme du type construit `formule`) et renvoie la chaîne de caractères indiquant comment un être humain lirait la formule.

```
let rec string_of_formule formule = match formule with
| Vrai            -> "Vrai"
| Faux            -> "Faux"
| Variable v      -> string_of_int v
| Non      p      -> "non " ^ string_of_formule p
| Et       (p, q) -> "(" ^ string_of_formule p ^ " et " ^ string_of_formule q ^ ")"
| Ou       (p, q) -> "(" ^ string_of_formule p ^ " ou " ^ string_of_formule q ^ ")"
| Implique (p, q) -> "(" ^ string_of_formule p ^ " implique " ^ string_of_formule q ^ ")"
;;
```

C'est une fonction récursive qui effectue une *analyse de cas* (**match ... with**) sur la formule. Cette analyse de cas est similaire à ce qu'on fait habituellement sur les listes (cas vide, cas avec un élément de tête et une liste de queue).

C'est *grosso modo* sur ce modèle que l'on écrira les fonctions (éventuellement des fonctions auxiliaires) prenant en argument une formule.

2 Sémantique : que signifie cette formule ? Évaluation booléenne

Si l'on dispose :

- d'une formule `p : formule`
- et d'une liste `f : bool list`, appelée *environnement*, associant à chaque variable de 0 à `list_length f - 1` une valeur booléenne,

alors on peut calculer la *valeur booléenne* associée à la formule p sous l'environnement l, de la façon suivante (certainement vue en cours) :

- la constante $\top$ (resp. $\bot$) est associée à la valeur booléenne **true** (resp. **false**)
- une variable v est associée à la valeur booléenne indiquée par l'environnement : la valeur de l'élément en position v de la liste f
- si P et Q sont des formules, alors la valeur booléenne associée à $\neg P$ (resp. $P \wedge Q$, $P \vee Q$) et à f est la négation (resp. la conjonction, la disjonction) booléenne de la valeur booléenne associée à Q (resp. à P et à Q) sous l'environnement f

On dit que l'on *évalue* la formule p sous l'environnement f.

En utilisant *uniquement* (mais éventuellement à plusieurs reprises) l'instruction **if** *booléen* **then** *terme_si_vrai* **else** *terme_si_faux*, discriminant suivant la valeur d'un booléen, écrire, sur le modèle de `string_of_formule`, une fonction :

```
evaluer : formule -> bool list -> bool
```

telle que evaluer p f évalue la formule p sous l'environnement f. On a alors, que (evaluer p) : bool list -> bool est la *fonction booléenne associée* à la formule p, car elle attend un argument : la liste f qui précise la valeur des variables booléennes.

Remarque : si b est un booléen, il est équivalent d'écrire $b =$ **true** et b.

[1] Cette structure est au programme de la seconde année de classes préparatoires.
[2] Ou, plus exactement *arbre de syntaxe abstraite* – encore au programme de seconde année

3 Preuve par évaluation booléenne

On appelle *tautologie* (resp. *contradiction*) une formule propositionnelle qui est vraie (resp. fausse) quelle que soit la valeur booléenne associée à ses variables.

Donner des exemples de tautologies sans variable, à une variable, à deux variables.

- Écrire une fonction :
  `borne_variable : formule -> int`
  telle que, si p est une formule, alors `borne_variable p` est le plus petit nombre strictement supérieur à toutes les variables de p. On dit alors que p a `borne_variable p` variables.
- Puis, une fonction :
  `tous_environnements : int -> bool list list`
  telle que, si n est un entier, alors `tous_environnements n` est la liste de tous les environnements possibles associant différents jeux de valeurs aux variables de 0 à n-1.
- Enfin, une fonction :
  `est_tautologie : formule -> bool`
  telle que pour toute formule p, `est_tautologie p` est vrai si et seulement si p est une tautologie. Quelle est la complexité (en nombre d'appels à `evaluer`) de la fonction `est_tautologie`, en fonction du nombre de variables de la formule ?

Modifier la fonction `est_tautologie` précédente pour écrire :

- une fonction :
  `contre_exemples : formule -> bool list list`
  telle que, pour toutes formule p, `contre_exemples p` est la liste des environnements rendant fausse la valeur de vérité de p
- une fonction :
  `equivalentes : formule -> formule -> bool`
  telle que, pour toutes formules p et q, `equivalentes p q` est vrai si et seulement si p et q sont *sémantiquement équivalentes*, i.e. ssi elles prennent la même valeur de vérité quelles que soient les valeurs booléennes de leurs variables. On dit aussi que p et q *ont la même sémantique*. Il existe une autre façon d'écrire cette fonction, utilisant directement est_tautologie sur une formule bien choisie. La programmer.

4 Modélisation de problèmes par des formules

Exprimer les problèmes suivants à l'aide de formules logiques et résoudre ces problèmes en appliquant à ces formules les algorithmes des parties 3 (preuve par évaluation booléenne) et 5 (résolution).

4.1 Le Club écossais[3]

Il existe en Écosse un club très fermé qui obéit aux règles suivantes :

- Tout membre non-écossais porte des chaussettes rouges
- Tout membre porte un kilt ou ne porte pas de chaussettes rouges
- Les membres mariés ne sortent pas le dimanche
- Un membre sort le dimanche si et seulement s'il est écossais
- Tout membre qui porte un kilt est écossais et marié
- Tout membre écossais porte un kilt

Montrer que ce club est si fermé qu'il ne peut accepter personne !

4.2 La Guerre des Écoles[4]

C'est l'histoire d'un polytechnicien, d'un normalien et d'un centralien qui promeuvent leurs Écoles respectives au Forum des Grandes Écoles de votre Lycée Louis-le-Grand. Pour attirer les élèves, chacun propose un baratin plus ou moins soutenable.

Le polytechnicien dit : « Le normalien ment. »

Le normalien dit : « Le centralien ment. »

Le centralien dit : « Le polytechnicien et le normalien mentent tous les deux. »

De ces trois Écoles, lesquelles sont dignes de confiance et lesquelles ne le sont pas ?

[3] Merci à Jean-Christophe Filliâtre
[4] D'après Vincent Le Ligeour

5 Preuve par raisonnement syntaxique : résolution

On peut chercher à prouver qu'une formule est une contradiction, sans avoir à évaluer la formule, mais uniquement en procédant à des transformations sur celles-ci.

5.1 La chasse à la négation

En utilisant la loi de De Morgan, montrer que toute formule est équivalente à une formule dont les négations ne portent que sur des variables. Écrire une fonction :

`chasse_non : formule -> formule`

qui « pousse les négations » au niveau des variables.

5.2 Formes normales conjonctives et disjonctives

On appelle *littéral* toute formule de la forme v ou $\neg v$, où v est une variable.

```
type litteral =
| Var    of int
| Nonvar of int
;;
```

On dit qu'une formule est *en forme normale conjonctive* si et seulement si elle est de la forme :

$$\bigwedge_c \bigvee_d l_{c,d}$$

où $(l_{c,d})$ est une famille de *littéraux*. Elle est en *forme normale disjonctive* si et seulement si elle est de la forme :

$$\bigvee_d \bigwedge_c l_{d,c}$$

où $(l_{d,c})$ est une famille de *littéraux*.

Par convention, une conjonction (resp. disjonction) vide est égale à la constante $\top$ (resp. $\bot$).

Dans les deux cas, de telles formules seront représentées efficacement uniquement par la famille de littéraux $(l_{c,d})$ ou $(l_{d,c})$ qui y apparaît : donc, par des listes de listes de littéraux.

Si p est une formule, on appelle *forme normale conjonctive* (resp. *disjonctive*) de p toute formule de même sémantique que p et qui soit en forme normale conjonctive (resp. disjonctive).

Écrire deux fonctions *mutuellement récursives* :

`fn_conj : formule -> litteral list list`

`fn_disj : formule -> litteral list list`

telles que, si p est une formule *supposée sans implications (cf. 6.2) et telle que les négations ne portent que sur des variables*, alors `fn_conj p`

(resp. `fn_disj p`) est la famille de littéraux définissant une forme normale conjonctive (resp. disjonctive) de la formule p.

Identifier un problème algorithmique qui peut survenir.

5.3 Résolution

On veut prouver qu'une formule p est une contradiction. L'idée est, à partir de cette formule, de créer d'autres formules r telles que si p est vraie, alors r est vraie aussi.

Soient p et p' deux disjonctions de littéraux et a une variable telles que $p = a \vee \bigvee_d l_d$et $p' = (\neg a) \vee \bigvee_d l'_d$ (à l'ordre près des littéraux). Montrer que si p et p' sont vraies, alors la formule $\bigvee_d l_d \vee \bigvee_d l'_d$ est vraie. On dit qu'on a effectué une *étape de résolution*.

Écrire ensuite une fonction :

`resolution : litteral list list -> litteral list list -> litteral list list`

qui à partir de deux listes de disjonctions de littéraux l et l', produit la liste de toutes les disjonctions de littéraux produites par une étape de résolution en choisissant p dans l et p' dans l'.

En déduire une fonction permettant de démontrer qu'une formule en forme normale conjonctive (représentée par sa famille de littéraux) est une contradiction, en effectuant des étapes de résolution jusqu'à ce que l'une d'elles produise la disjonction vide ($\bot$). Pourquoi ce processus termine-t-il ? Est-il complet (i.e. permet-il de détecter toute contradiction ?) Estimer grossièrement sa complexité.

6 Systèmes complets

On appelle *système complet* tout ensemble Σ de connecteurs logiques tels que toute fonction booléenne puisse être représentée par une formule utilisant ces connecteurs. C'est-à-dire que, pour toute fonction f : bool list -> bool , et pour tout entier positif n, il existe une formule p à n variables et n'utilisant que des connecteurs de Σ , telle que pour tout environnement l de n variables, la valeur de vérité de p sous l'environnement l est $f\,l$.

On verra dans le cours que l'identification de systèmes complets permet de limiter l'éventail de types de composants à utiliser dans un circuit logique.

Montrer que le système $\{\top, \bot, \wedge, \vee\}$ n'est pas complet, en exhibant une fonction non représentable par une formule avec ces seuls connecteurs.

6.1 Le système $\{\top, \bot, \neg, \wedge, \vee, \Rightarrow\}$ est complet

Écrire une fonction :

`formule_of_fonction : (bool list -> bool) -> int -> formule`

telle que, si n est un nombre et f une fonction booléenne, alors `formule_of_fonction f n` est une formule (de type formule, utilisant le vrai, le faux, la conjonction, la disjonction et l'implication) à n variables, correspondant à la fonction booléenne f considérée seulement sur les listes à n éléments.

6.2 Le système $\{\top, \bot, \neg, \wedge, \vee\}$ est complet

Soient deux formules p et q, écrire $p \Rightarrow q$ en fonction de p et de q mais en n'utilisant que les connecteurs $\neg$, $\vee$.

En déduire une fonction :

`supprimer_implication : formule -> formule`

qui remplace toute formule p en une formule de même sémantique mais qui n'utilise plus l'implication. Montrer qu'elle conserve la sémantique des formules (i.e. que, pour toute formule p, p et `supprimer_implication p` ont même sémantique). En déduire que le système $\{\top, \bot, \wedge, \vee\}$ est complet.

6.3 Les systèmes $\{\top, \neg, \wedge\}$ et $\{\bot, \neg, \vee\}$ sont complets

En utilisant les lois de De Morgan, écrire une fonction :

`supprimer_ou : formule -> formule`

(respectivement `supprimer_et : formule -> formule`)

qui remplace toute formule p en une formule de même sémantique mais qui n'utilise plus ni le faux, ni l'implication ni la disjonction (respectivement ni le vrai, ni l'implication ni la conjonction). Montrer que ces deux fonctions conservent la sémantique des formules. En déduire que les systèmes $\{\top, \neg, \wedge\}$ et $\{\bot, \neg, \vee\}$ sont complets.

Pourquoi les systèmes $\{\neg, \wedge\}$ et $\{\neg, \vee\}$ ne sont-ils pas complets *en toute rigueur* ? (Le jour du concours, néanmoins, on dira tout de même qu'ils sont complets, car c'est un point de détail.)