TP de Logique 1 (GLIN402)

Licence Informatique - Université Montpellier 2
M. Leclère - leclere@lirmm.fr

Résumé

L'objectif de ce TP est d'implanter toutes les notions définies sur les propositions. Il s'agit donc d'une part de définir des représentations pour les *connecteur, symbole propositionnel, propositions (fbf), ensemble de propositions, interprétation, ensemble d'interprétations, littéral, clause, forme clausale* et des fonctions de manipulation de ces représentations. D'autre part d'implanter les différentes méthodes de preuve définies en logique des propositions. L'énoncé est écrit pour le langage Caml mais ce TP peut être réalisé dans un autre langage fonctionnel interprété, auquel cas les représentations proposées dans ce sujet devront être adaptées. Le fichier `libtplog.ml` contenant les premières définitions de type ainsi que le fichier `tplogique.ml` qui est le fichier dans lequel vous réaliserez le tp doivent être récupérés sur l'espace pédagogique : `cours GLIN402` → `Documents et liens` → `TP` → `version_Caml`.

1 Représentation des propositions

Afin de faciliter la manipulation des propositions en Caml, nous utiliserons une représentation préfixée (fonctionnelle) des propositions : un type `fbf` est proposé dans le fichier `libtplog.ml` ainsi que différents accesseurs pour les valeurs de ce type. Les connecteurs et constantes logiques peuvent être vus comme des fonctions (ce sont en fait des "constructeurs Caml") sans paramètre, unaire ou binaire : on utilisera la notation suivante `VRAI, FAUX, NON, ET, OU, IMP, EQU`. Un symbole propositionnel est simplement une chaîne de caractères. On doit préfixer cette chaîne par le constructeur `SYMB` dans une formule bien formée.

Ainsi la constante `"p"` peut être considérée comme étant du type **symbole** propositionnel, tandis que la constante `SYMB "p"` est du type **fbf**. Avec cette représentation, les formules bien formées $\neg p$ et $p \to q$ seront représentées par les valeurs : `NON(SYMB "p")` et `IMP(SYMB "p", SYMB "q")` (cf. valeur des variables `unsymbole`, `uneformule`, `f` et `g` dans `tplogique.ml`).

Q 1 Définir les formules suivantes dans cette représentation Caml :

$$F1 = a \wedge b \leftrightarrow \neg a \vee b$$

$$F2 = \neg(a \wedge \neg b) \vee \neg(a \to b)$$

$$F3 = \neg(a \to a \vee b) \wedge \neg\neg(a \wedge (b \vee \neg c))$$

$$F4 = (\neg a \vee b \vee d) \wedge (\neg d \vee c) \wedge (c \vee a) \wedge (\neg c \vee b) \wedge (\neg c \vee \neg b) \wedge (\neg b \vee d)$$

En plus de la déclaration des types *symbole* et *fbf*, le fichier `libtplog.ml` fournit les fonctions suivantes de manipulation des valeurs du type *fbf* :

- `is_atom : fbf -> bool` qui renvoie *true* si la fbf donnée est un atome, *false* sinon.
- `is_symb : fbf -> bool` qui renvoie *true* si la fbf donnée est réduite à un symbole propositionnel, *false* sinon.
- `is_cst_vrai : fbf -> bool` qui renvoie *true* si la fbf donnée est réduite à la constante logique $\top$, *false* sinon.
- `is_cst_faux : fbf -> bool` qui renvoie *true* si la fbf donnée est réduite à la constante logique $\bot$, *false* sinon.
- `is_neg : fbf -> bool` qui renvoie *true* si la racine de la fbf donnée est le connecteur $\neg$, *false* sinon.
- `is_bin : fbf -> bool` qui renvoie *true* si la racine de la fbf donnée est un connecteur binaire, *false* sinon.
- `is_conj : fbf -> bool` qui renvoie *true* si la racine de la fbf donnée est le connecteur $\wedge$, *false* sinon.
- `is_disj : fbf -> bool` qui renvoie *true* si la racine de la fbf donnée est le connecteur $\vee$, *false* sinon.
- `is_impl : fbf -> bool` qui renvoie *true* si la racine de la fbf donnée est le connecteur $\to$, *false* sinon.
- `is_equi : fbf -> bool` qui renvoie *true* si la racine de la fbf donnée est le connecteur $\leftrightarrow$, *false* sinon.
- `symb : fbf -> symbole` qui retourne le symbole propositionnel racine de la fbf. **Cette fonction ne doit s'appliquer qu'à une formule réduite à un symbole propositionnel**.
- `ssfbf : int * fbf -> fbf` qui étant donné un entier i et une formule f, retourne la sous-formule gauche de f quand i vaut 1 et la sous-formule droite de f quand i vaut 2. **Cette fonction ne doit pas être utilisée quand i est différent de 1 ou 2, quand i vaut 1 et que f n'a pas de sous-formule propre, quand i vaut 2 et que f n'a pas de sous-formule droite.**

2 Analyse syntaxique des propositions

Q 2 Écrire la fonction `nbc` : $fbf \to int$ qui retourne le nombre de connecteurs d'une proposition (nombre d'occurrences).

Q 3 Écrire la fonction `prof` : $fbf \to int$ qui retourne la profondeur de l'arbre associé à la proposition donnée. Vous disposez de la fonction `maximum` : $int \times int \to int$.

Types ensembles : Caml dispose d'une notion de module (type + fonctions de manipulation du type) permettant entre autre de définir des types abstraits génériques. Le module générique SET doit être "instancié" en précisant : *(i)* le nom **EnsTelem** (commençant par une majuscule) du module créé, *(ii)* le type **Telem** des éléments qu'il contiendra, *(iii)* une fonction anonyme binaire `fun(e1,e2) ->`*expression qui renvoie true si e1 égal e2 et false sinon* de test d'égalité sur ces éléments. Ceci se fait de la façon suivante :

```
module EnsTelem = SET(
   struct
     type t = Telem
     let equal = fun(e1,e2) -> expression qui renvoie true si e1 égal e2 et false sinon
   end );;
```

Suite à une telle déclaration, le type ensemble ainsi créé a pour nom **EnsTelem.t** ; les valeurs de ce type sont représentées par des listes de Telem (la liste vide représentant l'ensemble vide) ; on dispose de plus des fonctions de manipulation suivantes (chaque fonction étant préfixée par le nom du module créé) :

- `EnsTelem.empty` est une constante désignant l'ensemble vide du type `EnsTelem.t`.
- `EnsTelem.is_empty : EnsTelem.t -> bool` qui renvoie *true* si l'ensemble est vide, *false* sinon.
- `EnsTelem.cardinal : EnsTelem.t -> int` qui renvoie le nombre d'éléments dans l'ensemble.
- `EnsTelem.member : Telem * EnsTelem.t -> bool` qui renvoie *true* si l'élément appartient à l'ensemble, *false* sinon.
- `EnsTelem.subset : EnsTelem.t * EnsTelem.t -> bool` qui renvoie *true* si le premier ensemble est inclus (ou égal) dans le second, *false* sinon.
- `EnsTelem.equal : EnsTelem.t * EnsTelem.t -> bool` qui renvoie *true* si les deux ensembles sont égaux, *false* sinon.
- `EnsTelem.add : Telem * EnsTelem.t -> EnsTelem.t` qui étant donné un élément e et un ensemble E retourne l'ensemble $\{e\} \cup E$.
- `EnsTelem.remove : Telem * EnsTelem.t -> EnsTelem.t` qui étant donné un élément e et un ensemble E retourne l'ensemble $E \setminus \{e\}$.
- `EnsTelem.choice : EnsTelem.t -> Telem` qui retourne un élément de l'ensemble. **Cette fonction ne doit pas être utilisée si l'ensemble est vide**.
- `EnsTelem.union : EnsTelem.t * EnsTelem.t -> EnsTelem.t` qui retourne l'union des deux ensembles.
- `EnsTelem.inter : EnsTelem.t * EnsTelem.t -> EnsTelem.t` qui retourne l'intersection des deux ensembles.
- `EnsTelem.diff : EnsTelem.t * EnsTelem.t -> EnsTelem.t` qui retourne la différence ensembliste entre le premier ensemble et le deuxième.

Exemple : Pour déclarer un type ensemble d'entiers, on instanciera le type SET de la façon suivante :

```
module EnsInt = SET(
   struct
     type t = int
     let equal = fun(i1,i2) -> i1=i2
   end );;
```

Le type ensemble d'entiers ainsi créé a pour nom **EnsInt.t** et l'expression `EnsInt.union([2;3],[5;2])` désigne alors l'ensemble `[3;5;2]` de type `EnsInt.t`.

Q 4 Écrire la fonction `ens_symb` : $fbf \to EnsSP.t$ qui retourne l'ensemble des symboles propositionnels d'une proposition donnée. Pour cela vous définirez un type ensemble de symboles **EnsSP.t** à l'aide de la déclaration :

```
module EnsSP = SET(
   struct
     type t = symbole
     let equal = fun(s1,s2) -> s1=s2
   end );;
```

3 Affichage (infixé) d'une formule (exercice optionnel)

Q 5 Écrire une fonction `afficher` qui prend en donnée une fbf, et retourne une chaîne de caractères représentant la formule sous forme infixée (affichage classique). Par exemple `afficher(IMP(SYMB "a", NON(ET(SYMB "b",NON( SYMB "c")))))` retournera la chaîne `"(a -> !(b ^ !c))"`. On pourra choisir ici le symbole ! pour le connecteur $\neg$, l'accent circonflexe ˆ pour le $\wedge$, la lettre `v` pour le $\vee$, le tiret-supérieur `->` pour le $\rightarrow$ et le inférieur-tiret-supérieur `<->` pour le $\leftrightarrow$. Vous disposez de la fonction `concatener` : $string \times string \rightarrow string$.

4 Définition d'une structure d'interprétation

On se donne un type énuméré **valVerite** qui contient les valeurs de vérité `Zero` et `Un`. On va représenter une interprétation par un ensemble de couples `(symbole, valVerite)`. Pour cela on déclare un type **coupleIntp** et on créé un module `Intp` instance de SET. Ainsi le type interprétation sera pour nous le type ensembliste **Intp.t**. On accède au premier élément d'un couple par la fonction `fst` et au deuxième élément par la fonction `snd`. Exemple : `fst("p",Zero)` retourne `"p"` et `snd("p",Zero)` retourne `Zero`. Les déclarations suivantes permettent la création de ces 3 types :

```
type valVerite = Zero | Un;;
type coupleIntp = symbole * valVerite;;
module Intp = SET(
   struct
     type t = coupleIntp
     let equal = fun(c1,c2) -> c1=c2
   end );;
```

Q 6 Définir en Caml les 3 interprétations $I1$, $I2$, $I3$ suivantes : $I1(a) = I1(c) = 1$ et $I1(b) = 0$, $I2(a) = I2(b) = I2(c) = 0$, $I3(a) = I3(b) = I3(c) = 1$.

Q 7 Écrire la fonction `int_symb` : $symbole \times Intp.t \rightarrow valVerite$ qui retourne la valeur d'interprétation d'un symbole propositionnel donné dans une interprétation donnée (on supposera que ce symbole propositionnel apparaît dans la structure d'interprétation).

Q 8 Écrire les fonctions d'interprétation des connecteurs et constantes logiques : `int_non` : $valVerite \rightarrow valVerite$, `int_et` : $valVerite \times valVerite \rightarrow valVerite$, `int_ou`, `int_imp`, `int_equ`, `int_vrai`, `int_faux`.

Q 9 Finalement, écrire la fonction `valv` : $fbf \times Intp.t \rightarrow valVerite$ qui calcule la valeur de vérité d'une formule f pour une interprétation i **complète** pour f.

5 Modèles, satisfiabilité et validité d'une proposition

Afin d'étudier les propriétés sémantiques des propositions, on se dote d'un type ensemble d'interprétations `EnsIntp.t`. Comme le type `Intp` est déjà un type ensemble, la création d'un ensemble d'ensemble se fait simplement par la commande :

```
module EnsIntp = SET(Intp);;
```

Q 10 Définir en Caml l'ensemble de toutes les interprétations des symboles propositionnels p et q.

Pour tester la satisfiabilité d'une proposition, il faut calculer l'ensemble de ses interprétations qui ne dépend que de l'ensemble des symboles propositionnels apparaissant dans la proposition.

Q 11 Écrire une fonction `ens_int` qui prend en donnée un ensemble de symboles propositionnels (`EnsSP`) et retourne l'ensemble de toutes les interprétations (`EnsIntp`) de ces symboles propositionnels. Lorsqu'il n'y a qu'un symbole propositionnel, il n'y a que 2 interprétations possibles. Si il y en a plus d'une, il est judicieux de calculer récursivement l'ensemble I des interprétations de tous les symboles sauf le premier, puis de prendre en compte le premier symbole en ajoutant à chaque interprétation de I l'interprétation du premier symbole (une fois à 0 et une fois à 1). Vous serez certainement amener à écrire une fonction intermédiaire `add_all` qui ajoute un `couplIntp` à toutes les interprétations d'un ensemble d'interprétation.

Q 12 Écrire un prédicat `est_un_modele` qui, étant donné une fbf et une interprétation, retourne *true* si l'interprétation est un modèle de la formule.

Q 13 Écrire une fonction `modele` qui retourne l'ensemble des modèles d'une fbf donnée.

Q 14 Écrire un prédicat `satisfiable` qui retourne *true* si et seulement si une fbf donnée est satisfiable. Tester votre prédicat sur les propositions $a, \neg a, (a \wedge b), ((a \wedge b) \wedge \neg a), F_1, F_2, F_3, F_4$.

Q 15 Écrire un prédicat `valide` qui retourne *true* si et seulement si une fbf donnée est valide. Tester votre prédicat sur les propositions $a, \neg a, (a \vee b), ((a \vee b) \vee \neg a), F_1, F_2, F_3, F_4$.

6 Extension des notions de modèle à des ensembles de propositions

On se dote d'un type ensemble de formules bien formées `Ensfbf` :

```
module Ensfbf = SET(
   struct
     type t = fbf
     let equal = fun(f1,f2) -> f1=f2
   end );;
```

Q 16 Écrire la fonction `ens_symb_ensfbf` : $2^{PROP} \rightarrow 2^{S}$ qui retourne l'ensemble des symboles propositionnels d'un ensemble de propositions donné, extension de la fonction `ens_symb` à des ensembles de formules.

Q 17 Écrire une fonction `modele_commun` qui retourne l'ensemble des modèles commun à un ensemble de propositions.

Q 18 Écrire un prédicat `contradictoire` qui retourne *true* si et seulement si un ensemble de propositions est contradictoire.

7 Equivalence et conséquence entre propositions

Remarque. Dans ce qui suit, vous proposerez plusieurs versions : une s'appuyant sur la définition initiale (à partir des modèles) des notions d'équivalence et conséquence logiques et l'autre s'appuyant sur les propriétés qui lient ces notions à celles de validité et satisfiabilité. Vous en profiterez alors pour vérifier expérimentalement les propriétés démontrées en cours.

Q 19 Écrire **deux** versions d'un prédicat `equivalente` qui teste si deux fbf données sont sémantiquement équivalentes (idem elles ont les mêmes valeurs de vérité pour toutes les interprétations). Faire des tests ! En particulier $((a \vee b) \vee \neg a) \equiv \neg((c \wedge d) \wedge \neg c)$.

Q 20 Écrire **trois** versions d'un prédicat `consequence2` qui étant donnée 2 propositions $F1$ et $F2$, retourne *true* si $F2$ est conséquence logique de $F1$. Vérifier que $a \models (a \vee b)$, $a \not\models (a \wedge b)$, $((a \vee b) \vee \neg a) \models \neg((c \wedge d) \wedge \neg c)$, $((a \wedge b) \wedge \neg a) \models (c \vee d)$.

Q 21 Étendre la fonction précédente (des trois manières possibles) à un prédicat `consequence` prenant en donnée un ensemble de formules $\{f_1, f_2, ... f_n\}$ et une fbf f et retournant true si f est conséquence logique de $f_1 ... f_n$ (c'est à dire $\{f_1, f_2, ... f_n\} \models f$). Tester s procédure en vérifiant si $\{a \wedge b, \neg a, b \rightarrow d\} \models c \rightarrow d$.

8 Mise sous forme conjonctive

Les 5 premières questions visent à fournir les transformations de fbf permettant un passage à la forme conjonctive. Pour ces fonctions, il faut raisonner sur l'arbre syntaxique associé à la formule. La 6[e] vise à fournir cette fonction. Attention, la 5[e] fonction (`distOu`) est particulièrement délicate. Ex. : `distOu(ET(ET(SYMB "a", NON(SYMB "b")), OU(SYMB "c", OU(NON(SYMB "d"),ET(SYMB "e",SYMB "f")))))` doit retourner `ET(ET(SYMB "a", NON(SYMB "b")), ET(OU(SYMB "c",OU(NON(SYMB "d"),SYMB "e")),OU(SYMB "c",OU(NON(SYMB "d"),SYMB "f"))))`.

Q 22 Écrire une fonction récursive `ote_equ` qui prend en paramètre une fbf et retourne une fbf logiquement équivalente qui ne contient pas de connecteur $\leftrightarrow$. Rappel : $(A \leftrightarrow B) \;\equiv\; ((A \rightarrow B) \wedge (B \rightarrow A))$

Q 23 Écrire une fonction récursive `ote_imp` qui prend en paramètre une fbf et retourne une fbf logiquement équivalente qui ne contient pas de connecteur $\rightarrow$. Rappel : $(A \rightarrow B) \quad \equiv \quad (\neg A \vee B)$

Q 24 Écrire une fonction récursive `ote_constante` qui prend en paramètre une fbf et retourne une fbf logiquement équivalente qui ne contient pas de constante logique. Rappel : $\top \quad \equiv \quad (\neg p \vee p)$ et $\perp \quad \equiv \quad (\neg p \wedge p)$

Q 25 Écrire une fonction récursive `red_neg` qui prend en paramètre une fbf ne contenant pas de connecteur $\leftrightarrow$ et $\rightarrow$ et retourne une fbf logiquement équivalente dont la négation ne porte que sur les symboles propositionnels. Rappel : $\neg\neg A \quad \equiv \quad A, \quad \neg(A \wedge B) \quad \equiv \quad (\neg A \vee \neg B), \quad \neg(A \vee B) \quad \equiv \quad (\neg A \wedge \neg B)$

Q 26 Écrire une fonction récursive `dist_ou` qui prend en paramètre une fbf composée de littéraux connectés par des $\wedge$ et $\vee$ et retourne une fbf logiquement équivalente sous forme conjonctive (i.e. conjonction de disjonctions de littéraux). Rappel : $(A \vee (B \wedge C)) \quad \equiv \quad ((A \vee B) \wedge (A \vee C))$

Q 27 Écrire alors la fonction `forme_conj` qui prend en paramètre une fbf quelconque et retourne une fbf logiquement équivalente sous forme conjonctive.

9 Forme clausale

On propose de représenter un littéral **litteral** par un couple `(signe, symbole)` (le type signe étant un type énuméré). On représente alors une clause (type `Clause.t`) comme un ensemble de littéraux et une forme clausale (type `EnsClause.t`) comme un ensemble de clauses. Les déclarations suivantes permettent la création de ces 4 types :

```
type signe = PLUS | MOINS ;;
type litteral = signe * symbole ;;
module Clause = SET(
   struct
     type t = litteral
     let equal = fun(l1,l2) -> l1=l2
   end );;
module EnsClause = SET(Clause);;
```

Q 28 Écrire une fonction récursive `trans_clause` qui prend en paramètre une fbf disjonction de littéraux et retourne la clause correspondante à cette fbf.

Q 29 Écrire une fonction récursive `trans_ens_clause` qui prend en paramètre une fbf sous forme conjonctive et retourne l'ensemble de clauses correspondant à cette fbf.

Q 30 Finalement, écrire une fonction `forme_clausale` qui prend en paramètre une fbf quelconque et retourne l'ensemble de clauses correspondant à sa forme clausale.

10 Méthodes de preuve

Il s'agit d'implanter les méthodes vues en cours. Il est préférable d'en implanter une correctement et complètement plutôt que d'essayer de toutes mal les faire ! On s'attachera en particulier à :

- implanter la méthode ;
- proposer des fonctions `satisfiable`, `valide` et `consequence` qui s'appuie sur la méthode ;
- se doter d'un jeu d'essais permettant de tester la méthode.

Q 31 Mettre en œuvre la méthode de résolution sur `EnsClause`.

Q 32 Mettre en œuvre Davis et Putnam sur `EnsClause`.

Q 33 Mettre en œuvre la méthode des Tableaux sur `Ensfbf`.

11 Application

Q 34 Modéliser un problème en logique des propositions et résolvez-le à l'aide d'une des 3 méthodes précédentes.