TP n° 1

Logique propositionnelle

Hugo SALOU MPI*

Dernière mise à jour le 18 janvier 2023

1 Syntaxe et sémantique

1.1 Syntaxe

1.

```
type formule =
  | Var   of string
  | And   of formule * formule
  | Or    of formule * formule
  | Imply of formule * formule
  | Equiv of formule * formule
  | Top
  | Bot
  | Not   of formule
```

Code 1 – Définition du type `formule` représentant les éléments de $\mathcal{F}$

2.

```
let rec print_formule (x: formule) : unit =
  match x with
  | Var(p) -> print_string p;
  | And(p,q) ->
      print_string "(";
      print_formule p;
      print_string "∧";
      print_formule q;
      print_string ")";
  | Or(p,q) ->
      print_string "(";
      print_formule p;
      print_string "∨";
      print_formule q;
      print_string ")";
  | Imply(p,q) ->
      print_string "(";
      print_formule p;
      print_string "→";
      print_formule q;
      print_string ")";
  | Equiv(p,q) ->
      print_string "(";
      print_formule p;
      print_string "↔";
      print_formule q;
      print_string ")";
  | Top -> print_string "⊤";
  | Bot -> print_string "⊥";
  | Not(p) ->
      print_string "(";
      print_string "¬";
      print_formule p;
      print_string ")"
```

Code 2 – Affichage du type `formule`

3.

```
let rec vars (x: formule) : string set =
  match x with
  | Var(p) -> Set.add p Set.empty
  | And(p,q) | Or(p,q) | Imply(p,q) | Equiv(p,q) -> Set.
      ↪ union (vars p) (vars q)
  | Top | Bot -> Set.empty
  | Not(p) -> vars p
```

Code 3 – Ensemble de variables d'une formule de type `formule`

1.2 Sémantique

4.

```
type env_prop = (string * bool) list
```

Code 4 – Définition du type env_prop représentant un environnement propositionnel

5.

```
let print_env_prop (e: env_prop): unit =
  let strings = List.map
    (fun (p, b) -> p ^ "↦" ^ (if b then "V" else "F"))
    e in
  let concatenated = List.fold_left (fun s p -> s ^ "," ^ p
      ↪ ) "" strings in
  print_string("{" ^ concatenated ^ "}")
```

Code 5 – Affichage du type env_prop

6.

```
exception Missing_Env

let rec interprete (f: formule) (e: env_prop) : bool =
  match f with
  | Var(p) ->
      let rec aux e =
        match e with
        | [] -> raise Missing_Env
        | (v,t)::_ when t -> true
        | _::q -> aux q
      in aux e
  | And(p,q) -> (interprete p e) && (interprete q e)
  | Or(p,q) -> (interprete p e) || (interprete q e)
  | Imply(p,q) -> interprete (Or(q, Not(p))) e
  | Equiv(p,q) -> (interprete p e) = (interprete q e)
  | Top -> true
  | Bot -> false
  | Not(p) -> not (interprete p e)
```

Code 6 – Interprétation d'une formule

7.

```
let rec all_envs (vars: string list): env_prop list =
  match vars with
  | x :: q -> let envs = all_envs q in
    let aux1 e = (x, true) :: e in
    let aux2 e = (x, false) :: e in
    (List.map aux1 envs) @ (List.map aux2 envs)
  | [] -> []
```

Code 7 – Génération des environnements propositionnels

8.

```
let sat (f: formule): env_prop =
  let envs = all_envs (vars f) in
  let envs_valides = List.filter (interprete f) envs in
  match envs_valides with
  | [] -> raise Unsat
  | x::_ -> x
```

Code 8 – Résolution du problème Sat

9.

```
let est_valide (f: formule): bool =
  let envs = all_envs (vars f) in
  List.for_all (interprete f) envs
```

Code 9 – Résolution du problème Valide

10.

```
let est_cons_semantique (f: formule) (g: formule): bool =
  let envs = all_envs (vars f) in
  let envs_f_valides = List.filter (interprete f) envs in
  List.for_all (interprete g) envs_f_valides
```

Code 10 – Vérification de "conséquence sémantique"

11.

```
let equiv (f: formule) (g: formule): bool =
  let envs = all_envs (vars f) in
  let envs_f_valides = List.filter (interprete f) envs in
  let envs_g_valides = List.filter (interprete g) envs in
  envs_f_valides = envs_g_valides
```

Code 11 – Vérification de “équivalence”

12.

```

  let envs_g_valides = List.filter (interprete g) envs in
  envs_f_valides = envs_g_valides
```

Code 12 – Calcul de modèles d’une formule

```
let models (f: formule): env_prop set =
  let envs = all_envs (vars f) in
  Set.of_list (List.filter (interprete f) envs)
```

Code 13 – Calcul de modèles d’une formule

2 Construction d’une formule à partir d’une fonction

1.

```
let formule_of_fct_bool (vars: string list) (f: fct_bool):
    ↪ formule =
  let envs = all_envs vars in
  let envs_valides = List.filter f envs in
  let rec gen_conj (rho: env_prop): formule =
    match rho with
    | [] -> Top
    | (p,b)::q -> if b then And(Var(p), gen_conj q)
                  else And(Not(Var(p)), gen_conj q)
  in
  let conjs = List.map gen_conj envs_valides in
  List.fold_left (fun x a -> Or(x, a)) Bot conjs
```

Code 14 – Déterminaison d’une formule dont l’interprétation est `f` sous forme FND

2.

```
let formule_of_fct_bool2 (vars: string list) (f: fct_bool):
    ↪  formule =
  let f' rho = not (f rho) in
  let h = formule_of_fct_bool vars f' in
  let rec convert_not (h: formule) =
    match h with
    | Or(a, b) -> And(convert_not a, convert_not b)
    | And(a, b) -> Or(convert_not a, convert_not b)
    | Not(a) -> convert_not a
    | Var(p) -> Not(Var(p))
    | Imply(a,b) -> convert_not (Or(b, Not(a)))
    | Equiv(a,b) -> convert_not (And(Imply(a, b), Imply(b,
        ↪ a)))
    | Top -> Bot
    | Bot -> Top
  in convert_not h
```

Code 15 – Déterminaison d’une formule dont l’interprétation est `f` sous forme FNC

3 Application de règles de réécriture