# Rapport de Projet de Programmation Fonctionnelle

Analyse de Tautologie dans les Formules Propositionnelles avec OCaml

$A \Rightarrow (B \Rightarrow (A \wedge B))$

| A | B | $A \wedge B$ | $B \Rightarrow (A \wedge B)$ | $A \Rightarrow (B \Rightarrow (A \wedge B))$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |

**Encadrant :** Christian Codognet
**Formation :** L2 Informatique
**Étudiant :** Ahmed Amine BENHAMMADA
**Numéro d'étudiant :** 12107981

19 avril 2024

# Table des matières

# 1 Présentation du Problème de Tautologie

Le problème de tautologie consiste à déterminer si une formule propositionnelle est vraie pour toutes les distributions de valeurs de vérité possibles. En d'autres termes, il s'agit de vérifier si une formule est une tautologie, ce qui signifie qu'elle est toujours vraie, quelle que soit la valeur de vérité attribuée à ses variables.

Dans ce problème, nous souhaitons développer un algorithme en OCaml pour vérifier si une formule propositionnelle donnée est une tautologie. Pour ce faire, nous utiliserons deux types de données différents : un type pour représenter les formules propositionnelles et un autre type pour les formules conditionnelles. Notre algorithme procèdera en trois étapes principales : la transformation de la formule propositionnelle en une forme équivalente utilisant des formules conditionnelles, la mise en forme normale de cette formule, et enfin, l'évaluation de la tautologie.

**Exemple** Considérons la formule propositionnelle suivante :

$$(p \wedge q) \vee (\neg p \wedge q)$$

Nous devons déterminer si cette formule est une tautologie. Pour ce faire, nous utiliserons notre algorithme en OCaml pour la transformer en une forme conditionnelle, puis nous la mettrons en forme normale et enfin nous évaluerons si elle est une tautologie.

## 1.1 Solution Algorithmique

Pour aborder et résoudre algorithmiquement le problème de vérification de tautologie pour des formules propositionnelles en utilisant les types `prop` et `cond` en OCaml, nous pouvons développer une solution en suivant les étapes de transformation, mise en forme normale, et évaluation. Voici une description plus détaillée de chacune de ces étapes avec des éléments algorithmiques :

### 1.1.1 Étape 1 : Transformation de `prop` en `cond`

Dans cette étape, nous convertissons une formule du type `prop` à une formule du type `cond` en utilisant uniquement le connecteur `Si`. Cela implique de réinterpréter chaque connecteur logique (`Et`, `Ou`, `Imp`, `Equiv`, `Non`) en termes de `Si`.

— `Non (p)` se traduit par `Si(p, Faux_bis, Vrai_bis)`.
— `Et (p1, p2)` se traduit par `Si(p1, p2, Faux_bis)`.
— `Ou (p1, p2)` se traduit par `Si(p1, Vrai_bis, p2)`.
— `Imp (p1, p2)` se traduit par `Si(p1, p2, Vrai_bis)`.
— `Equiv (p1, p2)` se traduit par `Si(p1, p2, Si(p2, Faux_bis, Vrai_bis))`.

Ces transformations permettent de n'utiliser que des expressions conditionnelles pour représenter toute logique propositionnelle.

### 1.1.2 Étape 2 : Mise en Forme Normale

L'objectif est de s'assurer que le premier argument du connecteur `Si` est toujours une variable ou une constante, jamais un autre `Si`. Pour cela, nous appliquons une transformation récursive pour remanier les imbrications de `Si` :

$$\texttt{Si(Si(a, b, c), d, e)} \rightarrow \texttt{Si(a, Si(b, d, e), Si(c, d, e))}$$

Cela garantit que les évaluations successives n'ont pas à gérer un `Si` en premier argument, simplifiant ainsi l'évaluation logique.

### 1.1.3 Étape 3 : Évaluation de Tautologie

Pour vérifier si une formule est une tautologie, nous devons évaluer la formule dans tous les contextes possibles de vérité des variables utilisées. L'évaluation utilise un environnement qui associe chaque variable à une valeur de vérité :

— L'évaluation de `Vrai_bis` retourne `true`.
— L'évaluation de `Faux_bis` retourne `false`.
— L'évaluation de `W(i)` retourne la valeur associée dans l'environnement ; si elle n'est pas présente, elle retourne `false` par défaut.
— L'évaluation de `Si(g, h, k)` évalue d'abord `g`. Si `g` est vrai, alors `h` est évalué ; sinon, `k` est évalué.

Pour vérifier si la formule transformée et normalisée est une tautologie, nous testons toutes les combinaisons de valeurs de vérité pour les variables et vérifions si le résultat est toujours vrai. Cela peut être réalisé par un balayage exhaustif des valeurs possibles des variables.

# 2 Implémentation

Pour implémenter l'algorithme de vérification de tautologie en OCaml, nous utilisons deux types de données pour représenter les formules propositionnelles et les conditions, ainsi que des fonctions pour effectuer la transformation, la mise en forme normale et l'évaluation.

## 2.1 Types

Nous définissons deux types de données :

— `prop` : Ce type représente les formules propositionnelles. Il peut être une variable (`V`), vrai (`Vrai`), faux (`Faux`), ou une combinaison de formules avec des connecteurs logiques (`Et`, `Ou`, `Imp`, `Equiv`, `Non`).

```
(* Définition des types pour les formules propositionnelles et conditionnelles *)
type prop =
  | V of int              (* Représente une variable avec un identifiant *)
  | Vrai                  (* Représente la constante vrai *)
  | Faux                  (* Représente la constante faux *)
  | Et of prop * prop     (* Représente la conjonction de deux propositions *)
  | Ou of prop * prop     (* Représente la disjonction de deux propositions *)
  | Imp of prop * prop    (* Représente l'implication entre deux propositions *)
  | Equiv of prop * prop  (* Représente l'équivalence entre deux propositions *)
  | Non of prop           (* Représente la négation d'une proposition *)
```

— `cond` : Ce type représente les conditions utilisées pour évaluer les formules. Il peut être une variable (`W`), vrai (`Vrai_bis`), faux (`Faux_bis`), ou une combinaison conditionnelle (`Si`) utilisant le connecteur `Si`.

```
type cond =
  (* Représente une variable dans le contexte des conditions *)
  | W of int
  (* Représente la constante vrai dans le contexte des conditions *)
  | Vrai_bis
```

```
  (* Représente la constante faux dans le contexte des conditions *)
  | Faux_bis
  (* Représente une conditionnelle avec une condition et deux branches *)
  | Si of cond * cond * cond
```

## 2.2 Algorithme

Voici l'algorithme complet avec des commentaires détaillés expliquant chaque partie du code :

### 2.2.1 Transformation de prop en cond

```
(* 1/Fonction de transformation *)
(* Convertit une formule propositionnelle en une formule conditionnelle *)
let rec trad_prop_to_cond = function
  (* Convertit une variable prop en variable cond *)
  | V(i) -> W(i)
  (* Convertit la constante vrai prop en constante vrai cond *)
  | Vrai -> Vrai_bis
  (* Convertit la constante faux prop en constante faux cond *)
  | Faux -> Faux_bis
  (* Convertit la négation d'une proposition prop en conditionnelle *)
  | Non(p) -> Si(trad_prop_to_cond p, Faux_bis, Vrai_bis)
  (* Convertit la conjonction de deux propositions prop en conditionnelle *)
  | Et(p1, p2) -> Si(trad_prop_to_cond p1, trad_prop_to_cond p2, Faux_bis)
  (* Convertit la disjonction de deux propositions prop en conditionnelle *)
  | Ou(p1, p2) -> Si(trad_prop_to_cond p1, Vrai_bis, trad_prop_to_cond p2)
  (* Convertit l'implication entre deux propositions prop en conditionnelle *)
  | Imp(p1, p2) -> Si(trad_prop_to_cond (Non p1), trad_prop_to_cond p2, Vrai_bis)
  (* Convertit l'équivalence entre deux propositions prop en conditionnelle *)
  | Equiv(p1, p2) -> Si(Si(trad_prop_to_cond p1, trad_prop_to_cond p2, Vrai_bis)
                    , Si(trad_prop_to_cond p2, trad_prop_to_cond p1, Vrai_bis), Vrai_bis)
```

### 2.2.2 Mise en forme normale

```
(* 2/Mettre une formule conditionnelle en forme normale *)
let rec form_norm_cond = function
  (* Si la formule est de la forme Si(Si(...), ...), on la décompose *)
  | Si(Si(a, b, c), d, e) ->
      (* Forme normale des parties conditionnelles *)
      let nb = form_norm_cond (Si(b, d, e)) in
      let nc = form_norm_cond (Si(c, d, e)) in
      (* Forme normale de la formule principale *)
      form_norm_cond (Si(form_norm_cond a, nb, nc))
```

```
  (* Si la formule est de la forme Si(...), forme normale de ses parties *)
  | Si(a, b, c) -> Si(a, form_norm_cond b, form_norm_cond c)
  (* Pour les autres cas, la formule est déjà en forme normale *)
  | other -> other
```

### 2.2.3 Évaluation dans un environnement

```
(* 3/Évalue une formule conditionnelle dans un environnement donné *)
let rec eval_cond f env =
  match f with
  (* Si la formule est vraie, retourne vrai *)
  | Vrai_bis -> true
  (* Si la formule est fausse, retourne faux *)
  | Faux_bis -> false
  (* Si la formule est une variable, recherche sa valeur dans l'environnement *)
  | W(i) -> List.assoc_opt i env |> Option.value ~default:false
  (* Si la formule est de la forme Si(...), évalue les parties conditionnelles *)
  | Si(g, h, k) ->
      (* Si la première partie est vraie, évalue la deuxième partie (branche alors *)
      if eval_cond g env then eval_cond h env
      (* Sinon, évalue la troisième partie (branche sinon) *)
      else eval_cond k env
```

### 2.2.4 Vérification de tautologie

```
(* 4/Fonctions de vérification du caractère de tautologie *)
(* Vérifie si une formule conditionnelle est une tautologie *)
let rec is_tautology_cond f =
  let rec try_all_assignments vars env =
    match vars with
    | [] -> eval_cond f env
    (* Sinon, pour chaque variable, essaie les deux valeurs possibles (vrai et faux) *)
    | v::vs ->
        try_all_assignments vs ((v, true)::env) && try_all_assignments vs ((v, false)::env)
  in
  let rec collect_vars f acc =
    match f with
    (* Si la formule est une variable, ajoute son identifiant à la liste des variables *)
    | W(i) -> if List.mem i acc then acc else i::acc
    (* Si la formule est de la forme Si(...), collecte les variables de ses parties *)
    | Si(g, h, k) -> collect_vars g (collect_vars h (collect_vars k acc))
    (* Pour les autres cas, aucune variable à collecter *)
    | _ -> acc
  in
```

```
  (* Collecte toutes les variables de la formule *)
  let vars = collect_vars f [] in
  (* Essaie toutes les affectations possibles *)
  try_all_assignments vars []
```

**Fonction principale pour le Toplevel :**

```
(* Fonction principale qui Vérifie si une formule propositionnelle est une tautologie *)
let is_tautology f =
  (* Convertit la formule propositionnelle en formule conditionnelle *)
  let g = trad_prop_to_cond f in
  (* Met la formule conditionnelle en forme normale *)
  let h = form_norm_cond g in
  (* Vérifie si la formule conditionnelle est une tautologie *)
  let is_tautology = is_tautology_cond h in
  let message = if is_tautology then "La formule est une tautologie."
                else "La formule n'est pas une tautologie." in
  (* Affiche le message indiquant si la formule est une tautologie *)
  print_endline message;
  (* Retourne la valeur booléenne (caractère de tautologie)
     indiquant si la formule est une tautologie *)
  is_tautology
```

Cet algorithme utilise les types `prop` et `cond` pour représenter les formules propositionnelles et les conditions, respectivement. Il effectue une transformation de la formule propositionnelle en une forme conditionnelle, puis la met en forme normale avant de l'évaluer pour déterminer si elle est une tautologie à l'aide de la fonction *is_tautologie*.

# 3 Tests et Jeux d'Essais

Pour vérifier la tautologie d'une *prop* j'ai établit une fonction `is_tautology` qui sert comme une fonction principale de test de tautologie , elle prend une prop à l'entrée qui est déjà transformée en *cond* , mise en forme normale et évaluée , et renvoie le caractère de tautologie; un boolean qui détermine si la formule est une tautologie ou pas. Ci-dessous quelques essais :

```
keshrud@keshrud:~$ ocaml
        OCaml version 4.13.1

# #use "tautologie.ml";;
type prop =
    V of int
  | Vrai
  | Faux
  | Et of prop * prop
  | Ou of prop * prop
```

```
  | Imp of prop * prop
  | Equiv of prop * prop
  | Non of prop
type cond = W of int | Vrai_bis | Faux_bis | Si of cond * cond * cond
val trad_prop_to_cond : prop -> cond = <fun>
val form_norm_cond : cond -> cond = <fun>
val eval_cond : cond -> (int * bool) list -> bool = <fun>
val is_tautology_cond : cond -> bool = <fun>
val is_tautology : prop -> bool = <fun>

# is_tautology Vrai;;
La formule est une tautologie.
- : bool = true

# is_tautology (Non(V(1)));;
La formule n'est pas une tautologie.
- : bool = false

# is_tautology (Et(Vrai, Ou(Non(V(1)), V(2))));;
La formule n'est pas une tautologie.
- : bool = false

# is_tautology (Non(Et(Vrai, Non(Et(V(1), V(2))))));;
La formule n'est pas une tautologie.
- : bool = false

# is_tautology (Imp(Et(Vrai, Ou(V(1), Non(V(2)))), Ou(Non(V(1)), V(2))));;
La formule est une tautologie.
- : bool = true

# is_tautology (Equiv(Et(Vrai, Non(Vrai)), Non(Vrai)));;
La formule est une tautologie : true
- : bool = true

# is_tautology (Equiv(Non(Et(V(1), Ou(V(2), Non(V(3))))),
                 Ou(Et(Non(V(1)), Non(V(2))), V(3))));;
La formule n'est pas une tautologie.
- : bool = false

# is_tautology (Imp(Et(Vrai, Ou(V(1), Non(V(2)))), Ou(Non(V(1)), V(2))));;
La formule est une tautologie.
- : bool = true

# is_tautology (Equiv(Et(V(1), Non(V(2))), Ou(Non(V(1)), V(2))));;
La formule n'est pas une tautologie.
- : bool = false
```

```
# is_tautology (Equiv(Et(V(1), Non(V(2))), Ou(Non(V(1)), Non(V(2)))));;
La formule n'est pas une tautologie.
- : bool = false

# is_tautology (Imp(Et(Vrai, Ou(V(1), Non(V(2)))), Ou(Non(V(1)), V(2))));;
La formule est une tautologie.
- : bool = true

# is_tautology (Equiv(Et(Et(V(1), Ou(V(2), Non(V(3)))), Non(Et(V(4), V(5)))),
                 Ou(Non(V(1)), Et(V(3), Non(V(4))))));;
La formule n'est pas une tautologie.
- : bool = false
```

# 4 Problèmes rencontrés et leurs solutions

Lors de l'implémentation de l'algorithme de vérification de tautologie en OCaml, plusieurs difficultés ont été rencontrées. Dans cette section, nous discutons des principaux problèmes rencontrés et des solutions adoptées pour les résoudre.

## 4.1 Utilisation des environnements OCaml

L'un des défis initiaux a été de comprendre et d'utiliser efficacement la notion d'environnement OCaml pour stocker les valeurs des variables propositionnelles. Cela impliquait de consulter plusieurs documentations OCaml pour comprendre les structures de données et les fonctions disponibles.

Pour résoudre ce problème, une approche progressive a été adoptée. En comprenant d'abord les types de données disponibles en OCaml, nous avons ensuite étudié les fonctionnalités des listes, qui ont été utilisées pour représenter les environnements. Voici un extrait du code illustrant l'utilisation des listes pour stocker les valeurs des variables :

```
1 (* Utilisation d'une liste pour stocker les valeurs des variables dans un environnement *)
2 let rec eval_cond f env =
3   match f with
4   | W(i) -> List.assoc_opt i env |> Option.value ~default:false
5   ...
```

## 4.2 Fonction OCaml try_all_assignments

Une autre difficulté était l'implémentation de la fonction `try_all_assignments`, qui teste toutes les affectations possibles des valeurs de vérité aux variables propositionnelles. Cette fonction était essentielle pour vérifier si une formule conditionnelle est une tautologie.

Après avoir consulté la documentation OCaml et étudié des exemples similaires, nous avons pu écrire la fonction `try_all_assignments` de manière efficace. Voici un exemple du code correspondant :

```
(* Fonction récursive pour tester toutes les affectations possibles *)
let rec try_all_assignments vars env =
  match vars with
  | [] -> eval_cond f env
  | v::vs ->
      try_all_assignments vs ((v, true)::env) && try_all_assignments vs ((v, false)::env)
```

# 5 ** CODE **

```
(* Définition des types pour les formules propositionnelles et conditionnelles *)
type prop =
  | V of int              (* Variable propositionnelle avec un identifiant *)
  | Vrai                  (* Constante "Vrai" *)
  | Faux                  (* Constante "Faux" *)
  | Et of prop * prop     (* Opérateur "Et" *)
  | Ou of prop * prop     (* Opérateur "Ou" *)
  | Imp of prop * prop    (* Opérateur "Implication" *)
  | Equiv of prop * prop  (* Opérateur "Équivalence" *)
  | Non of prop           (* Opérateur "Non" *)

type cond =
  | W of int              (* Variable conditionnelle avec un identifiant *)
  | Vrai_bis              (* Constante "Vrai_bis" *)
  | Faux_bis              (* Constante "Faux_bis" *)
  | Si of cond * cond * cond (* Opérateur conditionnel "Si" *)

(* 1/Fonction de transformation *)
(* Convertit une formule propositionnelle en une formule conditionnelle *)
let rec trad_prop_to_cond = function
  (* Convertit une variable prop en variable cond *)
  | V(i) -> W(i)
  (* Convertit la constante vrai prop en constante vrai cond *)
  | Vrai -> Vrai_bis
  (* Convertit la constante faux prop en constante faux cond *)
  | Faux -> Faux_bis
  (* Convertit la négation d'une proposition prop en conditionnelle *)
  | Non(p) -> Si(trad_prop_to_cond p, Faux_bis, Vrai_bis)
  (* Convertit la conjonction de deux propositions prop en conditionnelle *)
  | Et(p1, p2) -> Si(trad_prop_to_cond p1, trad_prop_to_cond p2, Faux_bis)
  (* Convertit la disjonction de deux propositions prop en conditionnelle *)
  | Ou(p1, p2) -> Si(trad_prop_to_cond p1, Vrai_bis, trad_prop_to_cond p2)
  (* Convertit l'implication entre deux propositions prop en conditionnelle *)
  | Imp(p1, p2) -> Si(trad_prop_to_cond (Non p1), trad_prop_to_cond p2, Vrai_bis)
  (* Convertit l'équivalence entre deux propositions prop en conditionnelle *)
```

```
  | Equiv(p1, p2) -> Si(Si(trad_prop_to_cond p1, trad_prop_to_cond p2, Vrai_bis), Si(trad_prop_to_cond

(* 2/Met une formule conditionnelle en forme normale *)
let rec form_norm_cond = function
  (* Si la formule est de la forme Si(Si(...), ...), on la décompose *)
  | Si(Si(a, b, c), d, e) ->
      (* Forme normale des parties conditionnelles *)
      let nb = form_norm_cond (Si(b, d, e)) in
      let nc = form_norm_cond (Si(c, d, e)) in
      (* Forme normale de la formule principale *)
      form_norm_cond (Si(form_norm_cond a, nb, nc))
  (* Si la formule est de la forme Si(...), forme normale de ses parties *)
  | Si(a, b, c) -> Si(a, form_norm_cond b, form_norm_cond c)
  (* Pour les autres cas, la formule est déjà en forme normale *)
  | other -> other

(* 3/Évalue une formule conditionnelle dans un environnement donné *)
let rec eval_cond f env =
  match f with
  (* Si la formule est vraie, retourne vrai *)
  | Vrai_bis -> true
  (* Si la formule est fausse, retourne faux *)
  | Faux_bis -> false
  (* Si la formule est une variable, recherche sa valeur dans l'environnement *)
  | W(i) -> List.assoc_opt i env |> Option.value ~default:false
  (* Si la formule est de la forme Si(...), évalue les parties conditionnelles *)
  | Si(g, h, k) ->
      (* Si la première partie est vraie, évalue la deuxième partie *)
      if eval_cond g env then eval_cond h env
      (* Sinon, évalue la troisième partie *)
      else eval_cond k env

(* 4/Fonctions de vérification du caractère de tautologie *)
(* Vérifie si une formule conditionnelle est une tautologie *)
let rec is_tautology_cond f =
  let rec try_all_assignments vars env =
    match vars with
    | [] -> eval_cond f env
    (* Sinon, pour chaque variable, essaie les deux valeurs possibles (vrai et faux) *)
    | v::vs ->
        try_all_assignments vs ((v, true)::env) && try_all_assignments vs ((v, false)::env)
  in
  let rec collect_vars f acc =
    match f with
    (* Si la formule est une variable, ajoute son identifiant à la liste des variables *)
    | W(i) -> if List.mem i acc then acc else i::acc
    (* Si la formule est de la forme Si(...), collecte les variables de ses parties *)
```

```
    | Si(g, h, k) -> collect_vars g (collect_vars h (collect_vars k acc))
    (* Pour les autres cas, aucune variable à collecter *)
    | _ -> acc
  in
  (* Collecte toutes les variables de la formule *)
  let vars = collect_vars f [] in
  (* Essaie toutes les affectations possibles *)
  try_all_assignments vars []

(* Fonction principale qui Vérifie si une formule propositionnelle est une tautologie *)
let is_tautology f =
  (* Convertit la formule propositionnelle en formule conditionnelle *)
  let g = trad_prop_to_cond f in
  (* Met la formule conditionnelle en forme normale *)
  let h = form_norm_cond g in
  (* Vérifie si la formule conditionnelle est une tautologie *)
  let is_tautology = is_tautology_cond h in
  let message = if is_tautology then "La formule est une tautologie."
                else "La formule n'est pas une tautologie." in
  (* Affiche le message indiquant si la formule est une tautologie *)
  print_endline message;
  (* Retourne la valeur booléenne (caractère de tautologie)
      indiquant si la formule est une tautologie *)
  is_tautology
```