UNIVERSITÉ SORBONNE PARIS NORD

Rapport du devoir de Programmation Fonctionnelle

L2 double-licence mathématiques et informatique

Par :

- Wilen YAICI 12214142
- Mohammed Yanis TAKBOU 12206365

Table des matières :

Introduction :

La programmation fonctionnelle et la logique partagent un lien profond, se nourrissant mutuellement pour résoudre des problèmes complexes de manière élégante et rigoureuse. La programmation fonctionnelle, en mettant l'accent sur les fonctions comme éléments fondamentaux du langage, adopte une approche déclarative qui favorise la clarté et la concision du code. De son côté, la logique offre un cadre formel pour exprimer et raisonner sur des propositions et des déductions, facilitant ainsi la résolution de problèmes complexes.

Dans ce projet, nous explorons cette intersection entre la programmation fonctionnelle et la logique, en cherchant à résoudre un défi qui demande la manipulation de concepts logiques à travers des techniques de programmation fonctionnelle. Nous nous appuierons sur les principes de la programmation fonctionnelle pour exprimer et évaluer des formules logiques, explorant ainsi la puissance de ce paradigme pour résoudre des problèmes de logique de manière élégante et efficace.

Description du problème :

Le problème abordé concerne la détermination de la tautologie d'une formule propositionnelle en logique. Une tautologie est une expression logique qui est toujours vraie, indépendamment des valeurs de vérité de ses variables propositionnelles.

La résolution de ce problème nécessite plusieurs étapes. Tout d'abord, la formule propositionnelle est traduite en une forme conditionnelle pour simplifier son évaluation. Ensuite, cette formule est transformée en forme normale pour une manipulation optimale. Enfin, en évaluant toutes les valeurs de vérité possibles pour ses variables, on détermine si la formule est toujours vraie. Si c'est le cas, elle est considérée comme une tautologie.

En résumé, le problème consiste à déterminer si une formule logique est une tautologie en évaluant sa validité pour toutes les valeurs possibles de ses propositions atomiques.

Solution algorithmique :

Pour résoudre le défi de déterminer si une formule propositionnelle est une tautologie, nous avons élaboré une approche algorithmique qui repose sur deux types distincts et implique trois étapes clés, conformément aux spécifications du projet. Les deux types utilisés sont les suivants : **prop** pour représenter les formules propositionnelles et **cond** pour les formules conditionnelles.

Pour garantir une transformation précise et cohérente des formules propositionnelles, nous avons élaboré trois étapes clés qui guident notre approche algorithmique

Étape 1 : Transformation d'une formule F de type `prop` en formule G type `cond`

Dans cette première étape, nous réalisons la transformation d'une formule propositionnelle de type `prop` en une formule logiquement équivalente de type `cond`, en respectant les règles suivantes :

- Chaque variable propositionnelle **`V(i)`** est traduite en une variable de type `cond` **`W(i)`**.
- Toute constante (**`Vrai`** ou **`Faux`**) reste inchangée.
- Tout connecteur logique présent dans le type `prop` est remplacé par le connecteur conditionnel "**`Si`**" dans le type `cond`.

La transformation est effectuée au moyen d'une fonction nommée `trad` de type `prop` -> `cond` . Cette fonction accepte une formule de type propositionnelle en tant qu'entrée et retourne une formule de type conditionnelle. Les détails de cette transformation seront explicités ultérieurement.

Étape 2 : Mise en forme normale de la formule G

Dans cette étape, nous nous engageons dans un processus de normalisation de la formule G, laquelle est exprimée selon le type `cond`. L'objectif est de la convertir en une forme équivalente H, toujours de type cond, mais présentée sous une forme dite "normale". Cette normalisation est caractérisée par une contrainte spécifique : le premier argument de toute occurrence de l'opérateur conditionnel `"Si"` doit être soit une variable, soit une constante, mais jamais une autre expression `"Si"`.

Pour réaliser cette transformation, nous utilisons une fonction nommée **for_nor**. Cette fonction prend en entrée une formule G de type `cond` et génère en sortie une formule H, également de type `cond`, mais normalisée selon les critères spécifiés.

Pour clarifier le processus de transformation, vous trouverez ci-dessous une explication détaillée du fonctionnement de `for_nor` et de ses implications sur la structure des formules transformées.

Étape 3 : Détermination de la tautologie de la formule H

Dans cette étape, notre objectif est de déterminer si la formule H est une tautologie en utilisant un environnement. Un environnement est une structure de données sous forme d'une liste de couples **(i, b)**, où `i` représente l'indice de la variable **W(i)** et **b** est sa valeur de vérité (true ou false en OCaml). Initialement, l'environnement est vide.

Pour réaliser cette évaluation, nous utilisons une fonction appelée **eval**. Cette fonction prend en entrée la formule **f** à évaluer et l'environnement **e**, et elle renvoie la valeur de vérité évaluée de la formule **f** dans l'environnement donné **e**.

Dans ce processus, chaque composante de la formule est soumise à une évaluation récursive, avec une attention particulière portée à chaque opération effectuée par la fonction **eval**. L'environnement fourni est pris en compte à chaque étape de l'évaluation. La fonction **eval** est conçue pour interpréter correctement la logique des opérateurs conditionnels, des variables et des valeurs booléennes, garantissant ainsi une évaluation précise de la formule dans le contexte spécifié par l'environnement.

Pour une compréhension approfondie de ce processus, les détails de la fonction eval et son interaction avec la formule et l'environnement sont explicités dans le code des fonctions ci-dessous.

Listing des types OCaml :

`prop` : Ce type représente une formule propositionnelle il peut être construit à partir de différentes combinaisons de connecteurs logiques, tels que : `Et` , `Ou` , `Imp` pour l'implication , `Equiv` pour l'équivalence, `Non` pour la négation , ainsi que des variables `V(i)` où `i` est son indice, et des constantes `Vrai` , `Faux.`

Sa définition en ocaml :

```
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
;;
```

`cond`: Ce type définit une formule conditionnelle utilisant exclusivement le connecteur Si. Les formules de ce type sont élaborées à partir du constructeur Si, comportant trois arguments qui représentent respectivement les parties conditionnelle, positive et négative de la formule. Ce type nous sera utile pour normaliser les formules à tester

Sa définition en OCaml :

```
type cond =
 | W of int
 | Vrai_bis
 | Faux_bis
```

```
  | Si of cond * cond * cond
;;
```

Listing des fonction OCaml :

trad (prop -> cond) : Pour la transformation d'une formule de type `prop` en une formule de type `cond`, la fonction gère tous les cas possibles.

```
let rec trad (f: prop) : cond =

 match f with

 | V(i) -> W(i)  (* Toute variable V(i) de type prop  se traduit en
                                                W(i) de type cond*)

 | Vrai -> Vrai_bis   (* Vrai est traduit en Vrai*)

 | Faux -> Faux_bis  (* Faux est traduit en Faux *)

 (* Traduction équivalente du reste des connecteurs comme spécifié
dans le projet *)

 | Ou(p, q) -> Si(trad p, Vrai_bis, trad q)

 | Et(p, q) -> Si(trad p, trad q, Faux_bis)

 | Imp(p, q) -> Si(trad p, trad q, Vrai_bis)

 | Equiv(p, q) -> Si(Si(trad p, trad q, Vrai_bis),

                     Si(trad q, trad p, Vrai_bis), Vrai_bis)

 | Non(p) -> Si(trad p, Faux_bis, Vrai_bis)

;;
```

for_nor (cond -> cond) : Réalise la mise en forme normale d'une formule conditionnelle G en une forme équivalente H. La mise en forme normale garantit que le premier argument de tout `Si` est soit une variable, soit une constante, mais jamais un `Si` imbriqué.

```
let rec for_nor (g: cond) : cond =
```

```
 match g with
 | Si(Si(a, b, c), d, e) ->
     for_nor (Si(a, Si(b, d, e), Si(c, d, e)))
 | Si(a, b, c) ->
     Si(for_nor a, for_nor b, for_nor c)
 | _ -> g;;
```

eval (cond -> (int * bool) list -> bool): Détermine si une formule conditionnelle **f** est une tautologie en utilisant un environnement **e**, représenté par une liste de couples **(i, b)** où **i** est l'indice de la variable **W(i)** et **b** est sa valeur de vérité (`true` ou `false`).

```
let rec eval (f: cond) (e: (int * bool) list) : bool =
 match f with
 | Vrai_bis -> true     (* Vrai_bis et Faux_bis sont resp. évalués en
true et false *)
 | Faux_bis -> false
 | W(i) -> (* Toute variable conditionnelle est associée à sa valeur
de vérité si elle est dans l'environnement, sinon elle est évaluée à
faux. *)
      (match List.assoc_opt i e with
       | Some b -> b
       | None -> false
      )
 | Si(g, h, k) ->
     match g with
     | Vrai_bis -> eval h e (* si g est vraie, évalue la partie si*)
     | Faux_bis -> eval k e (* si g est fausse, évalue la partie
sinon*)
     | W(i) -> (* si g est une variable : *)
         (match List.assoc_opt i e with
          | Some b -> if b then eval h e else eval k e (* si g est
dans l'environnement, vérifie sa valeur de verité et évalue
conformément.*)
          | None -> (* si g n'est pas dans l'environnement : *)
              let e1 = (i, true) :: e in
              let e2 = (i, false) :: e in
```

```
            eval h e1 && eval k e2
                (* crée deux autres environnement ou g prends deux
                valeurs booléennes différentes, et évalue la formule
                h dans les deux cas possibles *)
        )
    | Si(_,_,_) -> (*envoie une erreur dans le reste des cas.*)
        failwith "Error";;
```

si_tautologie (prop -> bool) : Cette fonction teste si une formule propositionnelle est une tautologie.

```
let si_tautologie (fp: prop) : bool =
 let  fc = trad fp  in (*traduction de la formule propositionnelle *)
 let fn = for_nor fc in (*normalisation de la formule conditionnelle*)
 eval fn [] ;; (*evaluation de la formule conditionnelle normalisée*)
```

Jeux d’essais :

Afin d'évaluer rigoureusement la performance de notre implémentation, nous avons entrepris des tests exhaustifs sur un ensemble varié de formules propositionnelles.

Pour ce faire, nous avons élaboré une fonction dédiée nommée **test**. Cette fonction prend en entrée une formule propositionnelle spécifique et procède à son évaluation. Elle affiche ensuite de manière explicite si la formule est considérée comme une tautologie ou non, offrant ainsi une évaluation claire et précise de son caractère logique.

```
let test () =
 let fp = formule_propostionnelle in (* formule
propositionnelle est à remplacer par la formule a tester *)
 let resultat = si_tautologie fp in
 Printf.printf "La formule entrée est une
 tautologie : %b\n" resultat (* affiche true si c'est une
tautologie, sinon affiche false *)
 ;;
 let ()= resultat ();;
```

Nous testerons alors, par exemple:

Faux

résultat attendu : *false*
résultat obtenu : *false*

Ou(V(1), Et(Vrai, Vrai))

résultat attendu : *true*
résultat obtenu : *true*

Ou(V(1), Non(V(1)))

résultat attendu : *true*
résultat obtenu : *true*

Ou(V(1), Et(Vrai, Vrai))

résultat attendu : *true*

résultat obtenu : *true*

Ou(Ou(Non(V(1)),V(2)),
Ou(V(1),V(2)))

résultat attendu : *true*

résultat obtenu : *true*

Equiv(Et(V(1), Ou(Vrai, Faux)), Non(Faux))

résultat attendu : *false*

résultat obtenu : *false*

Et(Vrai, Non(Faux))

résultat attendu : *true*

résultat obtenu : *true*

Ou(Et(V(1), Non(V(2))), Non(Et(V(3), V(4))))

résultat attendu : *false*

résultat obtenu : *false*

Le jeu d'essais comporte une variété de formules propositionnelles, allant des formules simples aux formules avec des variables. Chaque formule est accompagnée du résultat attendu et du résultat effectivement obtenu lors de l'évaluation du programme. Ces résultats confirment la validité du programme pour différentes configurations de formules propositionnelles.