

Rapport

Programmation Fonctionnelle

Sommaire:

Introduction	2
Problème et solution algorithmique	2
Etape 1	2
Etape 2	2
Etape 3	2
Listing des types et fonctions OCaml	3
Jeux d'essais	5
Conclusion	7

Introduction

Dans ce rapport, nous présentons un algorithme implémenté en OCaml permettant de déterminer si une formule propositionnelle est une tautologie, c'est-à-dire si elle est vraie pour toutes les distributions de valeur de vérité possibles. Nous détaillons, dans ce devoir, le problème, la solution algorithmique proposée ainsi que l'implémentation en OCaml.

Problème et solution algorithmique

Le problème consiste à vérifier si une formule propositionnelle est vraie pour toute valeur de vérité possibles. Pour résoudre ce problème, nous avons divisé l'algorithme en trois étapes:

- Etape 1: Transformation de la formule

Dans cette étape, nous transformons la formule propositionnelle initiale en une formule équivalente de type cond. Cette transformation se fait en remplaçant chaque connecteur logique par des constructeurs `Si` et des valeurs appropriées. Par exemple, la formule propositionnelle `Et (Non (Faux), Vrai)` après transformation donne la formule conditionnelle suivante **`Si (Si (Faux_bis, Faux_bis, Vrai_bis), Vrai_bis, Faux_bis)`**).

- Etape 2: Mise en forme normale

La formule résultante de l'étape précédente est ensuite mise en forme normale, où le premier argument de tout `Si` est soit une variable, soit une constante mais jamais un deuxième `Si`.

Cette étape est réalisée en réarrangeant les structures `Si` à l'aide d'un appel récursif.

En reprenant l'exemple précédent, la formule conditionnelle précédente nous renvoie **`Si (Faux_bis, Si (Faux_bis, Vrai_bis, Faux_bis), Si (Vrai_bis, Vrai_bis, Faux_bis))`**).

- Etape 3: Vérification de la tautologie

Enfin, nous déterminons à l'aide de la formule obtenue précédemment si cette dernière est une tautologie ou non en évaluant toutes les distributions de valeurs de vérité possibles.

`Si (Faux_bis, Si (Faux_bis, Vrai_bis, Faux_bis), Si (Vrai_bis, Vrai_bis, Faux_bis))`

`-> Si (Vrai_bis, Vrai_bis, Faux_bis) -> Vrai_bis`

Listing des types et fonctions OCaml

Voici les définitions des types `prop` et `cond` que nous avons utilisés pour résoudre ce problème.

```
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 ;;
```

Les différentes fonctions que nous avons implémentés:

- **let rec transforme f :**

Prend en paramètre une formule propositionnel et la transforme en formule conditionnel équivalente. Pour ce faire, elle doit de façon récursive, “traduire” les 5 connecteurs (`Ou`, `Non`, `Et`, `Implication` et `équivalence`) en connecteur `Si` tel que `Si (cond1, Vrai, Faux)`.

Non: `Non(f1) -> Si(transforme f1, Faux_bis, Vrai_bis)`
Ici, simple inversion de Vrai_bis et Faux_bis car le connecteur “Non” renvoie l’opposé de son contenu(exemple :Non (Faux) = Vrai).

Ou: `Ou(f1, f2) -> Si(transforme f1, Vrai_bis, transforme f2)`
Si f1 s’avère vrai, inutile de se préoccuper de f2 et renvoyons vrai. sinon, récursive sur f2

Et:
`Et(f1, f2) -> Si(transforme f1, transforme f2, Faux_bis)`
Si f1 s’avère faux, inutile de se préoccuper de f2 et renvoyons faux. sinon, récursive sur f2

Implication: `Imp(f1, f2) -> Si(transforme f1, transforme f2, Vrai_bis)`

Si f1 s'avère faux, l'implication donne vrai selon la table de vérité (Faux => b à pour résultat Faux peu importe le b). On déplace donc le vrai à droite afin que la formule ayant faux obtient malgré tout un vrai. Sinon, f1 est vrai et le résultat dépendra de f2.

Équivalent: $\text{Equiv}(f1, f2) \rightarrow \text{Si}(\text{Si}(\text{transforme } f1, \text{transforme } f2, \text{Vrai_bis}), \text{Si}(\text{transforme } f2, \text{transforme } f1, \text{Vrai_bis}), \text{Faux_bis})$
f1 <=> f2 correspond à : f1 => f2 et f2 <= f1. Il suffit donc de répéter le même paterne utilisé par l'implication en alternant la position de f1 et f2. Bien évidemment si la première implication est fausse, l'intérêt pour f2 devient caduque et faux devient le boolean renvoyé par défaut.

- let rec for_normale f :

Passage de la formule conditionnel en forme normale. Cette étape implique donc l'utilisation de la formule $\text{Si}(\text{Si}(a, b, c), d, e)$ en $\text{Si}(a, \text{Si}(b, d, e), \text{Si}(c, d, e))$

- let rec eval f e / let rec verif_couple e l

A partir d'ici, il suffit de parcourir la formule de façon récursive afin de déterminer la tautologie. (Exemple : $\text{Si}(b, g, d)$. Si g s'avère vrai, on recursive sur la formule g sinon d. Et si on tombe directement sur un constructeur comme `Vrai_bis` ou `Faux_bis`, on le renvoie)

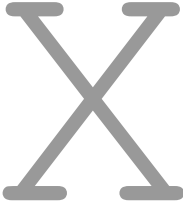
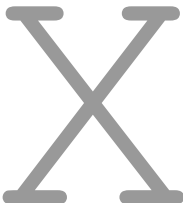
Comme la fonction utilise une notion d'environnement qui nécessite une liste, nous avons créé la fonction *verif_couple* qui prend en argument un entier e et une liste l (int * bool) et renvoie l'élément de la liste si e était présent à l'intérieur sinon liste vide (il n'était pas la)

Jeux d'essais

Nous avons effectué différents tests afin de tester la robustesse et l'efficacité de notre algorithme.

Voici les différentes formules que nous avons testées :

Propositionnelle	Conditionnelle	Normale	Evaluation
Imp (Non (Faux) , Ou (Non (Non (Faux)) , Non (V rai)))	(Si (Si Faux_bis, Faux_bis, Vrai_bis), (Si (Si (Si Faux_bis, Faux_bis, Vrai_bis), Faux_bis, Vrai_bis), Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis))), Vrai_bis)	(Si Faux_bis, (Si Faux_bis, (Si Faux_bis, (Si Faux_bis, Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis)), (Si Vrai_bis, Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis))), (Si Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis)), (Si Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis))), Vrai_bis), (Si Vrai_bis, (Si Faux_bis, (Si Faux_bis, (Si Faux_bis, Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis)), (Si Vrai_bis, Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis))), (Si	Faux (ce n'est pas une tautologie)

		Vrai_bis, (Si Faux_bis, Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis)), (Si Vrai_bis, Vrai_bis, (Si Vrai_bis, Faux_bis, Vrai_bis))), Vrai_bis))	
Et (Non (Faux), Vrai)	Si (Si (Faux_bis, Faux_bis, Vrai_bis), Vrai_bis, Faux_bis)	Si (Faux_bis, Si (Faux_bis, Vrai_bis, Faux_bis), Si (Vrai_bis, Vrai_bis, Faux_bis))	Vrai (c'est une tautologie)
	Si (Si (W 1, Vrai_bis, Faux_bis), Si (W 2, Vrai_bis, Faux_bis), Si (W 3, Vrai_bis, Faux_bis))	Si (W 1, Si (Vrai_bis, Si (W 2, Vrai_bis, Faux_bis), Si (W 3, Vrai_bis, Faux_bis))), Si (Faux_bis, Si (W 2, Vrai_bis, Faux_bis), Si (W 3, Vrai_bis, Faux_bis)))	Faux (ce n'est pas une tautologie)
	Si (Si (Si (W 1, Vrai_bis, Faux_bis), Si (W 2, Vrai_bis, Faux_bis), Si (W 3, Vrai_bis, Faux_bis))), Si (W 4, Vrai_bis, Faux_bis), Si (W 5, Vrai_bis, Faux_bis))	(Si W1, (Si Vrai_bis, (Si W2, (Si Vrai_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis))), (Si Faux_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis))), (Si W3, (Si Vrai_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5,	Faux (ce n'est pas une tautologie)

		<pre> Vrai_bis, Faux_bis)), (Si Faux_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis))), (Si Faux_bis, (Si W2, (Si Vrai_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis)), (Si Faux_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis))), (Si W3, (Si Vrai_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis)), (Si Faux_bis, (Si W4, Vrai_bis, Faux_bis), (Si W5, Vrai_bis, Faux_bis)))) </pre>	
--	--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--

Conclusion

En conclusion, ce devoir nous a permis de lier une notion que nous connaissions déjà à un langage de programmation fonctionnel tel que OCaml. En divisant ce problème en trois étapes distinctes, sa résolution est bien plus simple car nous avons une approche claire.

Lors de la première étape, il était nécessaire d'avoir une bonne compréhension des différents connecteurs logiques et de leurs équivalences dans les formules conditionnelles. Cela aide à mieux appréhender le devoir et avoir un bon départ. Ensuite, la mise en forme normale a permis de simplifier les structures conditionnelles, assurant que chaque structure `Si` ait comme premier argument soit une variable, soit une constante afin de faciliter la troisième étape.

Et pour finir la dernière étape, l'évaluation des formules afin de déterminer si elles sont des tautologies ou non, le cœur même de notre algorithme. Dans cette étape, nous avons évalué toutes les combinaisons possibles de ces valeurs de vérité.

Malgré l'achèvement de notre travail, nous avons rencontré quelques défis tout au long du processus. Des erreurs de syntaxe et de logique ont été inévitables, ce qui est normal dans tout projet de programmation. Cependant, une fois que nous avons identifié et compris ces erreurs, nous avons pu les éviter plus facilement dans les itérations suivantes de nos fonctions.

Parmi toutes les étapes, la troisième était un peu plus complexe à implémenter mais avec un peu de persévérance nous avons pu surmonter ces difficultés.

Ce projet nous a offert une expérience enrichissante en nous confrontant à quelques petites difficultés en programmation fonctionnelle. Avec une approche méthodique et persévérante, nous avons été en mesure d'avoir un algorithme fonctionnelle.

En continuant ainsi, nous sommes aussi en mesure de progresser et relever des défis plus défis dans d'autres matières.