Faculté des Sciences appliquées

INFO0054-1 Programmation Fonctionnelle

Projet : Tableaux sémantiques

Professeur :
Christophe DEBRUYNE

Assistant :
François ROZET

Groupe :
Guillaume DELPORTE
Maxime FIRRINCIELI
Arthur LOUIS

20 décembre 2021

Table des matières

1 Introduction

Dans ce projet, nous avons implémenté l'algorithme des tableaux sémantiques pour la logique propositionnelle. La logique propositionnelle est une branche des mathématiques étudiant les formules propositionnelles, relations logiques entre des variables connectées par des opérateurs logiques tels que :

— **ET**
— **OU**
— **NON**
— **SI** $\rightarrow$ **ENSUITE**
— ...

L'implémentation de cet algorithme et des règles d'élimination est appréhendée dans la première partie du rapport.

L'algorithme qu'il nous est demandé de mettre en place vise à simplifier l'écriture des tables de vérité de formules propositionnelles en éliminant les opérateurs logique afin d'obtenir des sous-formules contenant uniquement des variables ou leur complément. En parallèle à cet algorithme, nous avons également créé, comme demandé, une bibliothèque de fonctions permettant de manipuler les tableaux sémantiques obtenus et d'en apprendre plus sur les formules propositionnelles de base. Cette bibliothèque est décrite dans la seconde partie de ce rapport.

Enfin, il nous a été demandé d'améliorer notre algorithme afin d'implémenter des opérateurs logiques supplémentaires :

— **NON ET**
— **NON OU**
— **EXCLUSIF OU**
— **NON EXCLUSIF OU**
— **ÉQUIVALENT**

L'implémentation de ces opérateurs est explicitée dans la dernière partie de ce rapport.

2 Implémentation de `semtab`

Lors du début de ce projet, la grande question qui s'est avancée à nous était le choix de la structure de donnée à utiliser pour implémenter les formules et les tableaux afin de les traiter le plus facilement possible. Après réflexion, et hésitation avec l'utilisation d'arbres, nous nous sommes tournés vers les listes pour représenter les formules de base et des listes de sous-listes pour représenter le tableau sémantique et les formules développées.
Voici une brève explication du fonctionnement de l'algorithme :
En tout premier lieu, l'algorithme va regarder si la liste envoyée est vide. Si oui, il va alors renvoyer une liste vide.
L'algorithme va ensuite vérifier si tous les éléments de la formule sont développés grâce à une fonction `isDevelopped?` que nous avons implémentée spécifiquement dans ce but. Une formule totalement développée est une formule dans laquelle nous ne retrouvons plus d'opérateurs autres que l'opérateur **NON** et des variables seules.
Notre implémentation de l'algorithme `semtab` va ensuite récupérer la tête de liste et vérifier dans quel cas nous nous trouvons. Nous pouvons ainsi dénoter huit cas différents :

— Le **Cas de base** :
La tête de liste est une variable seule, dans ce cas, l'algorithme place la tête à la fin de la liste et se rappelle récursivement avec cette liste ainsi modifiée.

— Le double **NON** :
La tête de liste est composée d'un double **NOT**, l'algorithme se rappelle alors sur la même liste amputée de ces deux **NOT**.

— Le **ET** [1] :
La tête de liste est constituée de l'opérateur **AND**, `semtab` se rappelle alors sur une unique sous liste constituée des éléments du **AND**.

— le **NON ET** :
La tête de liste est composée d'un **NOT** qui entoure une condition **AND**, `semtab` divise alors la liste en deux sous-listes (comme dans un **OR**) avec dans chaque sous-liste, un **NOT** en tête car une condition **NOT(AND a b)** est vraie si a **OU** b est faux.

— le **OU**[1] :
La tête de liste est constituée de l'opérateur **OR**, dans ce cas, notre algorithme se rappelle deux fois sur deux sous listes qu'il crée composée du premier ou du deuxième membre du **OR** et du reste de la formule.

— le **NON OU** :
La tête de liste est composée d'un **NOT** qui entoure une condition **OU**, `semtab` divise alors la liste en une unique sous-liste (comme dans un **AND**) avec dans celle-ci, un **NOT** en tête car une condition **NOT(OR a b)** est vraie si a **ET** b sont faux.

— Le **SI** $\rightarrow$ **ENSUITE** :
La tête de liste est composée d'un **IFTHEN**, `semtab` divise alors la liste en deux sous-listes (comme dans un **OR**) avec dans une des deux sous-liste, un **NOT** en tête car une condition **(IFTHEN a b)** est vraie si a est faux **OU** b est vrai.

— Le **NON SI** $\rightarrow$ **ENSUITE** :
La tête de liste est composée d'un **NOT** qui entoure une condition **IFTHEN**, `semtab` réunit les deux éléments du **IFTHEN** avec devant le second élément du **IFTHEN** un **NOT** car une condition **NOT(IFTHEN a b)** est vraie si a est vrai **ET** b est faux.

L'algorithme va donc s'appeler récursivement en suivant ces cas spécifiques afin de développer les opérateurs restants. L'algorithme est ainsi prévu pour gérer tous les cas de base possibles. Nous avons également du adapter notre algorithme pour gérer les cas plus complexes avec des combinaisons d'opérateurs de base. Ainsi, par exemple, si l'on appelle notre fonction `semtab` sur une formule constituée de **NON** à la chaîne, celui-ci arrive bel et bien a renvoyer la formule de base simplifiée.

Voici quelques exemples d'utilisation de l'algorithme `semtab` [2] :

```
> (semtab '((AND a b))) ; exemple simple
(a b)
> (semtab '((OR a (AND b (NOT c))))) ; exemple de combinaison
((a) (b (NOT c)))
> (semtab '((NOT (NOT (NOT a))))) ; multiple NOT
((NOT a))
> (semtab '((NOT (OR a (AND b (AND c (IFTHEN (NOT d) e)))))) ; exemple complexe
(((NOT b) (NOT a)) (((NOT c) (NOT a)) ((NOT d) (NOT e) (NOT a))))
```

1. Cas réécris lors de l'extension de `semtab`, voir 4
2. Pour tous les tests de ce projet, merci d'utiliser la syntaxe proposée dans nos exemples, les opérateurs se retrouvent entre des parenthèses sinon certains tests risquent de ne pas fonctionner.

3 Implémentation de la bibliothèque

Pour la deuxième partie de ce projet, il nous a été demandé d'implémenter une bibliothèque de fonctions que nous pourrions utiliser afin de modifier, analyser, ou vérifier certaines propriétés sur nos tableaux sémantiques tout en nous basant sur la fonction `semtab` précédemment implémentée. C'est donc ce que nous avons fait pour toutes les fonctions constituant la bibliothèque demandées.

Voici de plus amples explications détaillées pour chaque fonction demandée faisant partie de la bibliothèque :

1. La fonction `satisfiable?` :
   Une liste de formules est dite satisfaisante si il y a au moins une des branches de tableau sémantique qui peut rendre la valeur **True**. C'est à dire qu'il est possible de construire au moins une table de vérité valide pour cette liste de formules.

   Pour réaliser ce test, nous introduisons le principe de branche et de contradiction dans une branche. Une branche est une sous-liste du tableau sémantique et ne contradiction dans une branche a lieu si dans une même branche on retrouve la variable **a** et **NOT a**. Nous réalisons une boucle sur chaque partie sur le tableau sémantique de la formule fournie à la fonction `satisfiable?` et vérifions si il y a ou non une contradiction dans la branche. Dès qu'une branche signale qu'elle n'a pas de contradiction, la fonction renvoie **True**. Ce test de contradiction est réalisé par la fonction `contradiction-in-branch?`.

   Pour tester notre fonction, il suffit d'appeler `satisfiable?` avec en argument la liste de formule souhaitée. Voici quelques exemples d'utilisation de la fonction :

   ```
   > (satisfiable? '((OR a (NOT a)))) ; exemple simple
   #t
   > (satisfiable? '((AND a (NOT a)))) ; cas simple
   #f
   > (satisfiable? '((AND a (OR b (IFTHEN (NOT a) c))))) ; exemple complexe
   #t
   ```

2. La fonction `valid?` :
   Une formule ψ [3] est dite valide sous un set d'hypothèses F si tous les modèles de F se retrouvent dans les modèles de ψ.

   Pour cela nous allons utiliser la fonctions `models` et plus précisément sa version étendue (tout cela est explicité dans la suite de cette section) pour comparer les modèles de la formule et des hypothèses. Si tous les modèles de F se retrouvent dans ψ, alors `valid?` renvoie **True**. Nous avons du faire attention à comparer les modèles élément par élément et non globalement car les variables dans un modèles peuvent être mélangées.

   Pour tester notre fonction, il suffit d'envoyer à `valid?` une formule (ou liste de formule) ψ dont on recherche la validité sous les hypothèses F. (`> (valid?` ψ F`)`). Voici quelques exemples d'utilisation de la fonction :

   ```
   > (valid? '(a b (NOT c)) '((OR a (NOT a)))) ; simple formule
   #f
   > (valid? '((OR a (AND b (NOT c)))) '((AND a (OR b c)))) ; liste de formules
   #t
   ```

3. Dans notre implémentation il est également possible d'envoyer une liste de formules car `semtab` est d'abord appliqué à la formule envoyée

3. La fonction `tautology?` :
Une tautologie est une formule qui est toujours vraie. Pour simplifier cette vérification, nous pouvons aussi dire que pour toute formule ψ soit une tautologie, **NOT** ψ doit être non-satisfaisante. C'est en suivant la deuxième définition que nous avons implémenté cette fonction. Celle-ci va donc tout simplement regarder si **chaque** branche de **NOT** ψ est **NOT** `satisfiable?`. Elle renverra **True** si c'est le cas et **False** dans le cas contraire.

   Pour tester notre fonction il suffit d'appeler `tautology?` sur une liste de formules. Voici quelques exemples d'utilisation de la fonction :

```
> (tautology? '((OR a (NOT a))))
#t
> (tautology? '((AND a (OR b c))))
#f
```

4. La fonction `contradiction?` :
Une contradiction est une formule qui est toujours fausse. Nous pouvons également dire que pour toute formule ψ soit une contradiction, celle-ci doit être non-satisfaisante. `contradiction?` renvoie **False** si il n'y a aucune contradiction dans ψ et renverra **True** sinon. Cette fonction va donc, pour une formule ψ donnée, regarder si ψ est **NOT** `satisfiable?`.

   Pour tester notre fonction, il suffit d'appeler `contradiction?` sur une liste de formule. Voici quelques exemples d'utilisation de la fonction :

```
> (contradiction? '((AND a (OR b (NOT a)))))
#f
> (contradiction? '((AND a (NOT a))))
#t
```

5. La fonction `models` :
Il est possible qu'une branche ouverte (sans contradiction) d'une liste de formules développée par `semtab` possède plus d'un modèle. En effet, si toutes les variables d'une liste de formule ne sont pas utilisées dans une branche, alors peut importe la valeur des variables non utilisées, la valeur de cette branche n'en dépendera pas. La fonction `models` vise à générer tout ces modèles. On introduit donc la notation suivante : $(\sim a)$ qui représente une variable non présente de base dans la branche et qui peut donc valoir a ou **NOT** a. On défini également la forme étendue de ces modèles qui étend $(\sim a)$ en développant la branche en deux sous-branches contenant pour l'une a et l'autre **NOT** a. La fonction `models` commence par vérifier que la liste de formules envoyée est `satisfiable?` car une liste de formules non-satisfaisante ne possède pas de modèles. Ensuite, la fonction retient les variables contenues dans la liste de formules de base et passe sur chaque branche ouverte pour en vérifier le contenu. Si dans une branche ouverte, il manque une ou plusieurs variables des formules de base, on rajoute autant de $(\sim \text{var})$ que nécessaire pour compléter le modèle.

   Pour utiliser notre fonction, il suffit d'appeler `models` avec la liste de formules souhaitée. Voici quelques exemples d'utilisation de la fonction :

```
> (models '((AND (NOT a) (OR b c)))) ; condensé
(((~ c) b (NOT a)) ((~ b) c (NOT a)))
> (models '((AND a (NOT a))))
Pas de modèles disponibles, la formule n'est pas satisfaisante
> (expands (models '((AND (NOT a) (OR b c))))) ; étendu
((b (NOT a) (NOT c)) (c (NOT a) b) (c (NOT a) (NOT b)))
```

6. La fonction `counterexamples` :
   Si une formule ψ[4] est non-valide sous une série d'hypothèses F c'est que des contre-exemples ont été trouvés entre les modèles de F et ψ (des modèles vrais pour F et non pour ψ. Notre fonction va commencer par vérifier si ψ est valide sous F. Si oui, aucun contre-exemple n'existe. Si non, on commence alors à comparer les modèles étendus de F et ψ. Tout modèle contenu dans F et non dans ψ est retourné afin d'obtenir la liste des contre-exemples. Encore une fois, nous avons eu besoin de faire attention à ne pas comparer des modèles dans leur ensemble car des éléments d'un modèle peuvent être mélangés.

   Pour tester notre fonction, il suffit d'envoyer à `counterexamples` une formule (ou liste de formule) ψ dont on recherche les contre-exemples sous les hypothèses F. (`> (counterexamples` ψ F`)`). Voici quelques exemples d'utilisation de la fonction :

```
> (counterexamples 'a '((OR a (NOT a)))) ; formule
(((NOT a)))
> (counterexamples '((OR a (AND b (NOT c)))) '((AND (NOT a) (OR b c)))) ; liste
((c (NOT a) b) (c (NOT a) (NOT b)))
> (counterexamples '((OR a (AND b (NOT c)))) '((AND a (OR b c))))
Pas de contre-exemples disponibles, formule valide sous hypothèse(s)
```

4 Extension de la fonction `semtab`

Dans cette dernière partie du projet, il nous est demandé d'étendre la fonction `semtab` afin que celle-ci puisse gérer les opérateurs spéciaux suivants :

— **EQUIVALENT**
— **OU EXCLUSIF**
— **NON ET**
— **NON OU EXCLUSIF**

Nous devions aussi proposer une généralisation pour que les opérateurs déjà présents **OU** et **ET** puissent comparer un nombre de variable(s) supérieur ou égal à un.
Dans un premier temps, commençons par définir les opérateurs supplémentaires. Pour nous aider définissons tout d'abord deux variables a et b que nous utiliseront dans des fins démonstratives. Continuons ensuite par définir ces opérateurs :

1. L'opérateur **EQUIV** :
   Deux variables a et b sont équivalentes si il est possible de déduire a à partir de b et inversement. a et b sont soit toutes les deux vraies, soit toutes les deux fausses. Ceci peut être modélisé à l'aide de la formule suivante, que nous avons alors implémentée :

$$\boxed{a \Leftrightarrow b \equiv (a \wedge b) \vee (\neg a \wedge \neg b)}$$

2. L'opérateur **XOR** :
   L'opérateur **OU EXCLUSIF** est vérifié si une seule des deux variables a ou b est vraie. Ceci peut être modélisé par la formule suivante que nous avons donc implémentée :

$$\boxed{a \oplus b \equiv (a \wedge \neg b) \vee (\neg a \wedge b)}$$

3. L'opérateur **NAND** :
   L'opération $a \barwedge b$ est vraie si a est faux ou si b est faux. Ceci peut être simplement représenté par la formule de logique propositionnelle suivante :

$$\boxed{a \barwedge b \equiv (\neg a \vee \neg b)}$$

4. Encore une fois il peut d'agir d'une liste de formules car nous appliquons `semtab` dessus

4. L'opérateur **XNOR** :
   Le **NON OU EXCLUSIF** renvoie exactement la même table de vérité que l'opérateur **EQUIVALENT**. Nous l'avons donc implémenté de la même façon :

$$\boxed{\neg(a \oplus b) \equiv (a \wedge b) \vee (\neg a \wedge \neg b)}$$

5. Extension de **OU** :
   Pour étendre l'opérateur **OU** vers $n \geq 1$ variable(s), nous avons utilisé la même règle d'élimination que précédemment, chaque membre du **OU** est isolé dans sa propre sous-liste à l'aide de la fonction `ORing`.

6. Extension de **ET** :
   Pour étendre l'opérateur **ET** vers $n \geq 1$ variable(s), nous avons utilisé la même règle d'élimination que précédemment, chaque membre du **ET** est regroupé dans la même sous-liste à l'aide de la fonction `ANDing`.