TP n°12 - Logique propositionnelle

Notions abordées

- Type récursif pour les formules propositionnelles
- Représentation des environnements par une liste d'associations
- Interprétation d'une formule selon un environnement
- Énumération de tous les environnements sur un ensemble de variables donné
- Pratique de Ocaml : fonctions du module List, type option, référence, exceptions

Toutes les fonctions doivent être commentées et **testées**. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.

Dans ce TP, on ne cherchera pas à rendre récursives terminales les fonctions récursives, sauf mention contraire explicite. De plus on écrira les définitions des types et des fonctions dans un fichier nommé `logique.ml` et les tests de ces fonctions dans un fichier à part nommé `test_logique.ml`. On pourra utiliser les fonctions du module `List` dont la documentation est disponible à l'adresse suivante. `https://ocaml.org/releases/4.12/api/List.html`

Exercice 0 Fonctions préliminaires

Dans ce TP on utilisera des listes pour représenter des ensembles, comme l'ensemble des variables par exemple. Afin d'éviter des représentations inutilement gourmandes en espace, il convient d'éviter les doublons dans ces listes.

Question 1

Définir une fonction `ajoute` qui prend en argument un élément `e` et une liste `l` calcule la liste obtenue en ajoutant à `l` l'élément `e` s'il n'y était pas déjà. *On veillera à la cohérence des types. De plus on pourra utiliser la fonction* `List.mem`.

Question 2

Définir une fonction `union` qui calcule une liste ayant pour ensemble d'élément l'union des ensembles d'éléments de chacune des listes.

Exercice 1 Modéliser les formules propositionnelles

Question 1

Définir le type `formule` pour modéliser les formules de la logique propositionnelle sachant qu'on modélisera une variable par une chaîne de caractères.

Question 2

Définir une fonction récursive `lvar` qui calcule une liste dont l'ensemble des éléments est l'ensemble des variables apparaissant dans une formule. *Dans un filtrage, il est possible de mettre plusieurs motifs avant une flèche, ce qui permet de factoriser l'écriture de la valeur de plusieurs cas distincts partageant la même valeur.*

Question 3

Pour faciliter le test de vos fonctions, le fichier `reader.ml` disponible sur cahier de prépa propose fournit un fonction `parse` qui prend en argument une chaîne de caractères et qui calcule l'objet de type `formule` que cette chaîne représente. Cela impose quelques contraintes sur l'écriture de cette chaîne :

- → les noms de variables sont constitués uniquement de lettres minuscules
- → la caractère `T` représente $\top$
- → la caractère `B` représente $\bot$
- → le caractère `~` représente $\neg$
- → le caractère `|` représente $\vee$
- → le caractère `&` représente $\wedge$
- → le caractère `>` représente $\rightarrow$
- → le caractère `=` représente $\leftrightarrow$

Par exemple `parse "a | (~ b) | T"` vaut `Disj (Disj (Var "a", Neg (Var "b")), Vrai)`, du moins pour une définition du type `formule` où le constructeur de la négation s'appelle `Neg`, celui de la disjonction s'appelle `Disj` et où le constructeur représentant $\top$ s'appelle `Vrai`.

Modifier le code de la fonction `reader` du fichier `reader.ml` pour le faire correspondre à la définition du type `formule` proposé à la question 1.

Question 4

Définir une fonction récursive `affiche_formule` qui affiche une formule. *On pourra suivre les conventions décrites pour les chaînes que la fonction* ***`parse`*** *prend en entrée.*

Exercice 2 Environnement et interprétation

On représente un environnement par une liste de couples de type `string * bool`, qui associent à chaque variable sa valeur de vérité. On considère donc la définition de type suivante.

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

On suppose qu'un environnement contient uniquement des couples dont les premières composantes sont 2 à 2 disjointes. Il faudra veiller à maintenir cette propriété. L'ordre des associations dans la liste est quelconque et non significatif. Deux fonctions outils sont données dans le fichier `outils_env.ml` disponible sur cahier de prépa afin de faciliter les tests :

→ `affiche_env` qui permet d'afficher les variables à vrai dans l'environnement ;

→ `cree_env` qui permet de créer un environnement à partir de la liste des variables et la liste de celles qu'on veut mettre à vrai.

De plus, la fonction `List.assoc` du module `List` pourrait être utile.

Question 1 💻

Définir une fonction `interprete` qui calcule l'interprétation d'une formule selon un environnement. Dans le cas où la formule contient une variable qui n'est pas présente dans l'environnement, la fonction lèvera une exception `Var_manquante` qui encapsule la variable manquante.

Exercice 3 Satisfiabilité et validité par énumération

Dans cet exercice on résout des problèmes sur les formules (comme décider si une formule est une tautologie, trouver un environnement satisfaisant une formule ou tester si une formule est conséquence logique d'une autre) en énumérant tous les environnements possibles. Pour ce faire, on construit à partir d'une liste de variables (2 à 2 distinctes), un générateur d'environnement. Plus précisément, un tel générateur est un fait la donnée d'une référence vers un environnement et d'une fonction qui renouvelle cet environnement, c'est-à-dire qui remplace l'environnement courant par un environnement nouveau, qui n'a pas déjà été proposé.

Question 1 ✍ 🪜

Lister tous les environnements possibles pour les variables `"a"`, `"b"`, et `"c"`.
D'une manière générale, pour n variables, combien y a-t-il d'environnements possibles ?

Question 2 ✍ 🪜

Rappeler l'algorithme d'incrémentation sur un nombre écrit en binaire. Itérer cet algorithme à partir du nombre écrit `000` pour obtenir les écritures en binaire des nombres de 0 à 7. Quel lien peut-on faire entre nombres et les environnements précédemment listés ?

Question 3 💻 (∗ mais nécessaire)

Définir une fonction `cree_generateur_env` qui prend en argument une liste de variables `lvar`, et calcule un couple `(e,next)` où `e` est une référence vers un environnement sur les variables de `lvar` dans lequel elles sont toutes à `false`, et où `next` est une fonction de type `unit -> unit` qui, lorsqu'elle est appelée, modifie l'environnement `!e` en un nouvel environnement si c'est possible, et qui lève l'exception `Fini` sinon. Plus précisément, le nouvel environnement doit être crée à partir de l'environnement courant (*i.e.* `!e`) en suivant l'algorithme d'incrémentation en binaire. On donne ci-après un exemple d'utilisation d'un tel générateur.

```
# let (e,next) = cree_generateur_env ["u";"v";"w"];;
val e : env ref = {contents = [("u", false); ("v", false); ("w", false)]}
val next : unit -> unit = <fun>

# !e;;
- : env = [("u", false); ("v", false); ("w", false)]

# next (); !e;;
- : env = [("u", true); ("v", false); ("w", false)]

# next (); !e;;
- : env = [("u", false); ("v", true); ("w", false)]

# next (); !e;;
- : env = [("u", true); ("v", true); ("w", false)]

# next ();next ();next ();next (); !e;;
- : env = [("u", true); ("v", true); ("w", true)]

# next ();
Exception: Fini.
```

Question 4 💻

Définir une fonction `est_sat_enum` de type `formule -> env option` qui calcule un environnement satisfaisant la formule prise en argument s'il en existe. *Plus précisément, la fonction calcule* `Some(e)` *où* `e` *est un environnement qui satisfait la formule si un tel environnement existe, et calcule* `None` *sinon.*

Question 5 💻

Définir une fonction `est_valide_enum` qui teste si une formule est valide, c'est-à-dire si c'est un tautologie. *On attend ici une valeur booléenne, on n'attend pas d'environnement ne satisfaisant pas la formule dans le cas où celle-ci n'est pas valide.*

Question 6 💻

Définir une fonction `est_csq_enum` qui teste si une formule est conséquence logique d'une autre.

Question 7 💻

Définir une fonction `sont_equiv` qui teste si deux formules sont équivalentes.

Exercice 4 Satisfiabilité et validité par l'algorithme de Quine

Dans cet exercice on s'intéresse à nouveau au problème de la satisfiabilité et de la validité d'une formule, mais on essaye d'éviter l'énumération des environnements en procédant par substitution et simplifications. Plus précisément, pour la satisfiabilité d'une formule A par exemple, il s'agit de substituer à une variable x la formule $\top$, de tester (récursivement) si la formule obtenue est satisfiable, auquel cas un environnement satisfaisant prolongé par $x \mapsto V$ satisfait A, et dans le cas contraire de tester (récursivement) si la formule obtenue en substituant à x la formule $\bot$ est satisfiable, auquel cas un environnement satisfaisant prolongé par $x \mapsto F$ satisfait A, et dans le cas contraire on conclut que la formule n'est pas satisfiable.

Question 1

Définir une fonction `substitue` qui prend en argument une formule `f`, une variable `var` et une autre formule `fvar` et qui calcule la formule obtenue en substituant dans `f` la formule `fvar` à toutes les occurrences de la variable `var`.

Question 2

Définir une fonction `simpl` qui simplifie une formule en appliquant là où elle le peut les règles de simplifications mentionnées en cours. On constate par exemple les affichages suivants (où les valeurs de retour de type `unit` ont été volontairement omises).

```
# affiche_formule (simpl (parse "(T|a) & (T|b)"));;
(T & T)

# affiche_formule (simpl (parse "T & T"));;
T

# affiche_formule (simpl (parse "(T|a) > b"));;
(T > b)

# affiche_formule (simpl (parse "T > b"));;
b

# affiche_formule (simpl (parse "b"));;
b
```

Question 3

Définir une fonction `simpl_complet` qui itère la fonction `simpl` jusqu'à ce que celle-ci n'ait plus d'effet. Par exemple, `simpl_complet (parse "(T|a) > b")` calcule `Var "b"`.

Question 4

Définir une fonction `est_sat_quine` de type `formule -> env option` qui calcule un environnement satisfaisant la formule prise en argument s'il en existe.

Question 5

Définir une fonction `est_non_sat_quine` de type `formule -> env option` qui calcule quant à elle un environnement **ne** satisfaisant **pas** la formule prise en argument s'il en existe.