DIAGRAMME DE DÉCISION BINAIRE ET TETRAVEX

Mai 2018

CALDAS Eduardo
SILVA DE OLIVEIRA Vinicius

Table des matières

1 Les Formules

Le module PropositionalLogic définit tout ce qu'on a besoin pour le traitement et manipulation des formules, il est bien sûr un functor, puisque le variables pourraient avoir des multiples types (int, char, string).

1.1 La Définition

D'abord nous définissons un type pour les formules :
type formule =
| Boolean of bool
| Variable of VT.t
| Bin_Op of formule $*$ (formule $\rightarrow$ formule $\rightarrow$ formule) $*$ formule
| Un_op of (formule $\rightarrow$ formule) $*$ formule

Nous avons choisi de définir des opérateurs le plus généraux possible, puisqu'ils partageaient plusieurs caractéristiques. Ensuite nous avons créé les différent constructeurs pour chaque opérateur.

On a donc une représentation d'une formule comme une arbre. Sa création sera décrite dans la prochaine section

1.2 L'évaluation d'une formule

Par la suite nous avons défini des fonctions pour évaluer ces formules :

- eval_one : Prends une variable "v" et une formule "f" en entré e calcule les deux valuations possibles de "f" par "v"
- eval_v : Prends une valuation "v, b" et une formule et choisis la bonne entre les deux possibilités données par eval_one
- eval_list : Applique eval_v à une liste

La création d'une fonction eval_one se justifie par la suite, lors de la création de la BDT.

1.3 La création des formules

La création, i.e. l'analyse syntaxique et lexicale des formules propositionnelles a été faite en deux parties.

1.3.1 Lexing

Le but de cette partie était de transformer l'entrée de façon a avoir une donnée plus régulière constitué de tokens. D'abord nous avons fait une régularisation de la string en entrée en utilisant des expression régulières :

- suppression des espaces
- transformations des chaînes encodantes, e.g. "<->", "true" en chaînes d'un caractère, "=", "T"

Dans la suite nous avons transformé la chaîne traité en tokens, en traitant les variables, qui peuvent être de plusieurs types (int, char, string)

1.3.2 Parsing

Maintenant on a une liste de tokens et l'on veut l'analyser syntaxiquement, ou interpréter dans une formule.

Pour ce faire on a d'abord écrit la grammaire en BNF(Backus-Naur Form) et en suite on a interpréter chaque type de token en suivant cette grammaire :

1. interpret_atoms : "T", "F", "var" , (<expr>) → Boolean, Variable, interpret <expr>
2. interpret_neg : <expr> → Neg (interpret_neg<expr>)
3. interpret_op : <expr> op <expr> → (interpret <expr>) op (interpret <expr>). En suivant l'ordre de precedence et l'associativité

Puisque le code comptait avec 100 lignes, et était assez complexe nous avons créé un tester contenant plus de 40 cas de test.

2 La BDD

2.1 La BDT

Nous avons commencé par créer la BDT à partir de la formule. On l'a définit comme suit :
type binary_Dtree =
| Leaf of bool
| Node of binary_Dtree * VT.t * binary_tree

2.2 Création de la BDT

En utilisant la fonction eval_one la création de la BDT à partir d'une formule était compacte. On prends comme entrée la formule "f" et l'ordre d'évaluation des variables "lv". L'algorithme procède récursivement, s'arrêtant lors d'une feuille :

1. (fFalse, fTrue) = eval_one lv.head f
2. Node (create_BDT fFalse lv.tail, lv.head, create_BDT fTrue lv.tail)

2.3 Compression BDT

Pour comprimer la BDT on a utilisé un Hashtbl pour enregistrer chaque nœud et après si l'on l'avait vu avant on utilise celui dans le Hashtbl

2.4 Satisfiabilité et Validité

La validité et satisfiabilité ont été faites avec un DFS récursif plus détaillés dans le code

3 Tetravex

3.1 Introduction et Stratégie

La résolution de Puzzle Tetravex est un problème de l'informatique NP-complet et peut être approché par différentes méthodologies. Dans le cadre de ce cours, après l'implémentation des BDD's et des proposition logiques, nous allons utiliser une approche pour cette problématique en utilisant la théorie de formules et BDD's développées et étudiées précédemment. La stratégie pour la résolution du Puzzle sera de transformer le tetravex en une formule et imposer qu'il y ait une résolution du tetravex si et seulement si la formule est satisfiable. Évidemment, on implémentera des BDD's correspondant pour étudier la satisfiabilité Pour cela, l'équipe était convaincu qu'une nouvelle définition, qui sera présentée dans les prochaines sections, pourrait être plus efficace que la compression fait avec l'arbre pour les formules et logiques propositionnelles vue avant.

Supposons que "q" est le numéro de carrés disponibles pour la résolution du tetravex et n, p les dimensions de la grille du Puzzle. On va créer n . p . c variables de la forme

$$\boxed{"a, b : c"} \tag{3.1}$$

$$a \in [1, n] b \in [1, p] c \in [1, q] \tag{3.2}$$

qui sera True si le carré c est dans la position (a, b) dans la grille et False sinon.

Notre "origine" de comptage sera en haut et à gauche avec les cordonnées (1, 1). La stratégie est basée sur la création de différentes formules qui seront combinées en sachant que chaque formule représente une règle du jeu (même numéro de chaque côté, limites du tableau, etc).

3.2 Modules et structure

Pour la résolution du Tetravex, nous avons créée les modules principaux suivants dans le file tetravex.ml :

- Foncteur : GeneralBDDTetravex (VT : Variable)
- Module Tetravex

3.2.1 Module-Foncteur GeneralBDDTetravex

C'est dans ce module qu'on va créer les relations entre variables et établir les règles du jeu à travers des BDD's des formules. Le code est bien documenté, mais on souligne ici les points principaux.

- type bdd C'est le type basique des bdd's. Il peut avoir un constructeur True, False ou un BddNode qui contiendra une tuple (id, type VT, bdd gauche, bdd droite)
- HTBDDdatabase est une Hashtable utilisé pour garder tous les bdd's déjà crées en forme d'un noeud avec "(b, a ; c)" comme valeur pour permettre un accès rapide aux arbres et éviter les créations inutiles

— HTRelationabetweenBDDsDatabases est une Hashtable utilisée pour comparer les différentes résultats d'opérations entre les différentes bdd's. L'objectif est surtout d'économiser des opérations avec cette Hashtable. On aura un container pour chaque opération suivante (
— notBDDTetravex prend une BDD et retourne la BDD avec la valeur boolean opposée utilisé le container "notContainer" pour garder les résultats
— andBDDTetravex prend deux BDD's et retourne une BDD qui représente la valeur de l'opération AND entre les bdd's
— orBDDTetravex prend deux BDD's et retourne une BDD qui représente la valeur de l'opération OR entre les bdd's
— bddLeftImpliesbddRight prend deux bdd's et les opérations logiques d'implication
— satisfact est la fonction qui permet de définir si la BDD est satisfaisable

3.2.2 Module Tetravex

Après la définition de nos bases, outils et opérations pour résoudre le Tetravex, on passe à la définition des formules à partir de variables et définitions faites auparavant. Dans ce module on définit deux classes : La classe carre qui représente les unités de carrés à être mis dans la grille. La classe tetravex qui contient les paramètres de la grille, les méthodes de création de formules et la liste de carre's disponibles. On va maintenant passer sur les méthodes principales de cette classe pour voir comment on traduit le Tetravex dans un problème de BDD :

— obligeRightCarre-to-belong et obligeLeftCarre-to-belong sont des méthodes qui retournent la bdd de possibilité d'un certain carre (carreAnalysed) à la position (a, b). Ceci correspond à la création de la formule pour la règle de même numéros côte-à-côte
— Le méthode uniquelyPlaceIn (carreAnalysed, a, b) est le méthode qui correspond à la règle de pouvoir de voir qu'un carré est placé à un seule placement. Dans ce cas là il y va voir si carré analysed est qu'en (a, b)
— Le méthode anyOtherCarreInThisGridPositionButCarreAnalised (carreAnalised : carre) a b Ce méthode va définir s'il n y a plus d'un carré par placement de grille
— conditionsCheckToPlaceAnalisedCarreInAB est le méthode qui assure les conditions de placement du carreAnalisé en une certaine position (a,b). Il retourne la bdd résultant.
— conditionsCheckToPlaceAnalisedCarreForAllPositions Ce méthode va analiser la possibilité de placement d'un carré spécifique pour toutes les positions.
— toutPlacer est le méthode qui assure la bonne mise en place des carrés dans la grille en suivant les règles définies avant
— existenceTetravex Une fois tout placé, ce méthode va checker s'il existe une conformation en voyant si les variables de chaque place de la grille a un carré correspondant
— solve est le méthode qui applique tous les autres méthodes pour afficher la solution. Il est très important de remarquer que ce méthode va mettre ensemble la condition de toutPlacer (le bon/correcte remplissage de la grille c'est à dire en suivant les règles) et l'existence, qui assure que toute la grille est remplie.

4 Testes et utilisation

L'utilisation suit les instructions du projet et quelques testes ont étés mis à disposition. La seule différence par rapport à la demande est que au lieu de ./bdd commande..., il faudrait mettre ./bdd.native commade...

La version d'OCaml utilisé était 4.06.1