LINFO1104: Projet du cours de paradigmes

Hadrien Allegaert & Jerome Lechat

April 23, 2021

1 Introduction

Ce projet avait pour but d'implémenter un arbre de décision et un mini jeu "Qui-est-ce?" selon une liste de record "ListOfCharacters" chargée à partir d'une base de donnée en format .txt.
Cette liste une fois chargée devait être tansformée en arbre de décision, permettant de retrouver le personnage choisi en répondant aux questions composant les noeuds de l'arbre construit.
Pour ce projet, nous avons choisi de n'utiliser que les techniques vues au cours, le code source ne contient donc pas de boucles à proprement parlé mais plutôt des fonctions récursives qui font office de boucles spécialisées.

2 Techniques utilisées

La solution devant être entièrement déclarative, aucune cellule, aucun objet ou type de donnée abstraits n'ont été utilisées. A la place, nous avons opté pour la construction d'un arbre de décision basé sur des records imbriqués.
Voici donc sa sémantique:

<DecisionTree> ::= leaf(<List>) | question(1:<atom> true:<DecisionTree>
false:<DecisionTree>).

Cette syntaxe permet d'avoir une définition récursive simpliste de l'arbre, et donc d'implémenter un algorithme récursif qui implémentera cette logique.

Comme spécifié dans l'introduction, pour ce projet nous n'utilisons pas de syntaxe de boucle "While" ou "For", nous utilisons des invariants de programmation et des fonctions récursives faisant appel à d'autres fonctions récursives de façon a exécuter la même séquence d'instruction qu'une boucle spécialisée.

Exemple avec la fonction QList permettant de calculer la différence de réponse "true/false" pour toutes les questions et retourne une liste de celle-ci : si 3 personnages ont tous les mêmes réponses, leur liste de différences sera une liste de 0.

Ce code se trouve dans le code source du projet ou sur GitHub, dans le fichier TreeBuilderFun.oz aux lignes 19 à 53.

3 Algorithmes sans extension

3.1 TreeBuilder

La fonction TreeBuilder est une fonction qui peut paraître assez complexe au premier abord. Mais en réalité, celle-ci n'est composé que de fonctions spécialisées, et dédiés à une tâche bien spécifique.
Cet algorithme peut se découper en sous-algorithmes:

- {QList L Q}
- {Builder L K D}
- {MaximumArity L Max ArityMax} uniquement pour l'extension "Incertitude base de donnée"

QList permet simplement de calculer la liste des différences de réponses type "vrai/faux" (en valeur absolue) à partir d'une liste de records (de personnages et leurs caractéristiques) et d'une liste de questions, sous forme d'atome.

Builder est une fonction lazy et un algorithme un peu plus complexe, qui lui prend en argument la liste de record (L), la liste de questions (K), la liste de différences (D) et qui enfin retourne l'arbre selon les questions ayant les différences absolues les plus petites. Le fait que ça soit une fonction lazy est voulu car si le joueur ne se trompe pas dans ses réponses on est en droit de supposés qu'à chaque étapes seul un sous arbres sera utile au résultat final.

3.2 GameDriver

Le GameDriver est une fonction assez simple finalement.

Se basant sur un pattern matching basique de la structure de l'arbre, cet algorithme se contente de poser la question contenue dans le noeud courant de l'arbre passé en argument grâce à la fonction ProjectLib.ask.
En fonction de la réponse de l'utilisateur (true ou false), l'algorithme se relance récursivement sur le sous-arbre contenu dans la clé (true ou false selon la réponse).
Et ainsi de suite jusqu'à tomber sur le pattern 'leaf(<list>)' qui est le cas de base de cet algorithme récursif.

A ce moment précis le GameDriver fait un appel à ProjectLib.found de façon à récupérer la liste de personnage en résultat et la print grâce à la fonction PrintList.

3.3 Code source

```
fun {GameDriver Tree}
    local Result in
        case Tree
            of nil then {ProjectLib.surrender}
            [] leaf(_|_) then
                {ProjectLib.found Tree.1 Result}
                if Result == false then {ProjectLib.surrender}
                else
                    case Result
                        of _|_ then {PrintList Result}
                        else {PrintList Result|nil}
                    end
                end
                unit % must return unit (project request)
            [] tree(q:Q true:T false:F) then
                local Ans = {ProjectLib.askQuestion Q} in
                    if Ans == true then {GameDriver T}
                    else {GameDriver F}
                end
        end
    end
end
```

Figure 1: GameDriver sans extension

3.3.1 Complexité du GameDriver

Le GameDriver de base a une complexité maximale de $\mathcal{O}(n) + \Theta(x)$ où n est la hauteur de l'arbre et x est le nombre de nom de personnages dans la liste finale à afficher. La complexité de $\Theta(x)$ vient de la fonction PrintList qui a une complexité égal à la longueur de la liste passée en arguments.
Et le GameDriver en lui-même peut selon l'arbre prendre jusqu'à n étapes pour trouver la bonne réponse (liste de personnage), n étant donc le nombre de noeuds entre la racine de l'arbre et sa feuille la plus éloignée (celle pour laquelle n est le plus grand).
On peut donc considérer sa complexité en $\mathcal{O}(n + x)$

4 Extensions choisies

Pour les quatres étoiles requises nous avons choisi d'implémenter les extensions suivantes:

- Incertitude sur la base de donnée ⋆
- Bouton "oups" (permet de revenir en arrière) ⋆⋆⋆

4.1 Choix d'implémentation

4.1.1 Incertitude sur la base de donnée

Pour cette extension, nous avons simplement changé deux lignes de code dans deux fonctions internes du TreeBuilder.

Initialement, celui-ci fait appel à un Spliter de liste permettant de définir à partir d'une sous-liste de personnages et d'une question deux nouvelles sous-listes : l'une contenant les personnages y répondant oui et l'autre y répondant non.

Nous avons donc ajouté un "try-catch" autour de la vérification de la réponse à une question car si une question n'est pas présente, ozengine nous renverra une erreur, si l'on répère cette erreur grâce au catch, vu qu'on ne peut pas choisir de sous-liste dans laquelle mettre notre pesonnage, nous l'ajoutons simplement dans les deux listes.

```
fun {Spliter L Q LTrue LFalse}
    /*
    Split the list L about question Q answer true or false
    if the answer of question Q is unknow then character is append in both lists
    */
    case L
        of nil then r(trueList:LTrue falseList:LFalse)
        [] H|T then
            if H.Q == true then {Spliter T Q H|LTrue LFalse}
            else {Spliter T Q LTrue H|LFalse}
            end
    end
end
```

Figure 2: Spliter sans extension

```
fun {Spliter L Q LTrue LFalse}
    /*
    Split the list L about question Q answer true or false
    if the answer of question Q is unknow then character is append in both lists
    */
    case L
        of nil then r(trueList:LTrue falseList:LFalse)
        [] H|T then
            try
                if H.Q == true then {Spliter T Q H|LTrue LFalse}
                else {Spliter T Q LTrue H|LFalse}
                end
            catch _ then
                {Spliter T Q H|LTrue H|LFalse} % gestion of incertitude on db
            end
    end
end
```

Figure 3: Spliter avec extension

La même logique a été applique à la fonction qui calcule la liste des différences de réponses true ou false (voir figure 1).

4.1.2 Bouton oups

Pour l'implémentation de cette extension, nous n'avons pas dû changer le TreeBuilder mais bien ajouter une fonction et un identifiant au GameDriver: SearchNode et TreeSave.
TreeSave va sauvegarder l'arbre de base (complet) dans une variable qui lui sera liée.
Ensuite, nous avons donc refactorisé le GameDriver de base sous une fonction interne appellée "Inner", de façon à ce qu'elle ne modifie pas, mais aie un accès dans son scope à la valeur de TreeSave.
Une fois cette modification légère effectuée, nous avons pu nous attaquer à la technique de recherche de la question précédente, et c'est ce que fait la fonction SearchNode prenant en parametre la sauvegarde de l'arbre complet et son sous-arbre actuel, Cette fonction va simplement chercher dans l'arbre complet le noeud pour lequel un de ses sous-arbres est notre sous-arbre actuel et retourner le sous-arbre affilié à ce noeud. C'est-à-dire par exemple:
Nous avons un arbre complet tel que :

```
tree(q:Q1
    true:tree(q:Q2 true:[Name1, Name2] false:tree(...))
    false:tree(q:Q3 true:[Name3, Name4] false:tree(...))
)
```

Si on est par exemple sur la question Q3 actuellement et que le joueur clique sur le bouton oops, la fonction va retourner l'arbre complet, et ceci est aussi valable si la question est Q2.
Si l'arbre actuel est le sous-arbre de la clé false de Q2 alors la fonction retournera le sous-arbre:

```
tree(q:Q2 true:[Name1, Name2] false:tree(...))
```

A partir de là, on relancera donc la fonction Inner du GameDriver avec ce sous-arbre renvoyé par notre fonction SearchNode.

```
fun {SearchNode Tree Current}
    case Tree
    of _|_ then false
    [] tree(q:Q true:T false:F) then
        if Current == T orelse Current == F then Tree
        else
            local X in
                X = {SearchNode T Current}
                case X
                of tree(q:_ true:_ false:_) then X
                else {SearchNode F Current}
                end
            end
        end
    end
end
```

Figure 4: Search node function

```
fun {Inner Tree}
    local Result in
        case Tree
        of nil then {ProjectLib.surrender}
        [] leaf(_|_) then
            {ProjectLib.found Tree.1 Result}
            if Result == false then {ProjectLib.surrender}
            else
                case Result
                of _|_ then {PrintList Result}
                else {PrintList Result|nil}
                end
            end
            unit % must return unit (project request)
        [] tree(q:Q true:T false:F) then
            local Ans = {ProjectLib.askQuestion Q} in
                if Ans == true then {Inner T}
                elseif Ans == false then {Inner F}
                % here TODO gestion of unknow answer
                elseif Ans == oops then
                    {Inner {SearchNode TreeSave Tree}}
                end
            end
        end
    end
end
TreeSave = Tree
{Inner Tree}
```

Figure 5: Fonction Inner + les deux seules instructions de GameDriver