Rapport de projet
Programmation fonctionnelle

Lilian MALLARDEAU

Semestre 2

Table des matières

Préambule

Le sujet du projet portait sur les formules de la logique propositionnelle. Une formule logique (ou proposision) est une affirmation qui peut être vraie ou fausse : on lui associe donc sa valeur de vérité, V (vrai) ou F (faux).

Les proposisions sont formées avec des variables logiques (qui peuvent également être vraies ou fausses), et des connecteurs (ou $\vee$, et $\wedge$, ...).

On s'intéresse plus particulièrement dans le sujet à une représentation des tables de vérités des formules logiques par des arbres. D'abord, on représente la formule par un arbre binaire, dans lequel chaque nœud correspond à une variable et chaque sous arbre (respectivement de droite ou de gauche) correspond au sous arbre de décision de la formule en remplaçant la variable du nœud par vrai ou faux.

Ensuite, on s'intéresse au *Binary Decision Diagram*, l'idée étant de simplifier notre arbre de décision, d'abord en repérant les redondances et en les éliminant, puis en simplifiant les nœuds qui ont le même successeur.

Enfin, on utilisera Dot pour générer des graphiques de nos arbres.

Arbres de décision

Afin de pouvoir manipuler aisément les formules logiques en Caml, on définit tout d'abord un type `tformula` :

```
type tformula =
| Value of bool (* ⊥ ou ⊤ *)
| Var of string (* Variable *)
| Not of tformula (* Negation *)
| And of tformula * tformula (* Conjonction *)
| Or of tformula * tformula (* Disjonction *)
| Implies of tformula * tformula (* Implication *)
| Equivalent of tformula * tformula (* Equivalence *)
```

Ainsi, une formule est soit une valeur de vérité (vrai ou faux) ou une variable, soit une composée d'autres formules et opérateurs ($\wedge$, $\vee$, $\neg$, ...).

On définit également un type `decTree` pour représenter un abre de décision, en consirérant qu'un arbre est soit une feuille (vrai ou faux), soit un nœud qui possède un nom (le nom de la variable correspondant au nœud) et ses deux sous arbres :

```
type decTree =
| DecLeaf of bool
| DecRoot of string * decTree * decTree
```

Question 1 : `getVars : tformula -> string list`

La première question consiste à écrire une fonction `getVars`, qui sur la donnée d'une formule nous renvoit la liste des variables présentes dans la formule, triée par ordre croissant et sans doublon.

Il a suffit pour cela de définir une fonction auxiliaire récursive, puis d'appliquer `List.sort_uniq compare` pour trier la liste en s'assurant que chaque élément n'y apparaisse qu'une fois (`compare` étant l'ordre naturel sur les chaînes de caractères).

Question 2 : `evalFormula : env -> tformula -> bool`

Afin de pouvoir évaluer une formule, on doit connaître la valeur des variables apparaissant dans la formule : on définit pour cela un **environnement** comment étant une liste de couples `(nom_variable, valeur)` :

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

On peut maintenant définir une fonction récursive `evalFormula`, qui prend en paramètres une formule et un environnement, et qui évalue cette formule dans cet environnement.

Pour cela, j'ai défini une fonction `getVarValue : env -> string -> bool`, qui prend en paramètres un environnement et le nom d'une variable, et renvoit la valeur de la variable dans l'environnement.

La fonction `evalFormula` se réduit donc à un pattern matching sur la formule, en appliquant les règles de logique pour déterminer la valeur de vérité de la formule en fonction des opérateurs, et en appelant la fonction `getVarValue` lorsqu'une formule est de type `Var(_)`.

Question 3 : `buildDecTree : tformula -> decTree`

On définit la fonction `buildDecTree : tformula -> decTree` qui construit l'arbre de décision d'une formule.

Pour construire l'arbre de décision, on doit construire l'environnement au fur et à mesure de la construction de l'arbre : pour cela, j'ai défini une fonction auxiliaire, qui prend l'environnement en tant qu'accumulateur, et la liste des variables qui n'ont pas encore été ajoutées à l'environnement. Lorsque toutes les variables ont été ajoutées à l'environnement, on peut évaluer la formule (et renvoyer un `DecLeaf`), sinon on fait deux appels récursifs, en ajoutant respectivement la valeur `true` et `false` pour la variable actuelle dans l'environnement :

```
let rec aux formula env vars = match vars with
  | [] -> DecLeaf (evalFormula env formula)
  | h::t -> DecRoot(h, aux formula ((h,true)::env) t,
     aux formula ((h,false)::env) t)
```

Il faut ensuite appeler la fonction auxiliaire avec un accumulateur vide et la liste des variables présentes dans la formule :

```
aux formula [] (getVars formula);;
```

Binary Decision Diagram

On cherche maintenant non plus à constuire un arbre, mais un graphe (*Binary Decision Diagram*), dont le chemin choisi définit les valeurs de chaque variable, et l'extrémité du chemin la valeur de la formule calculée avec l'environnement défini par le chemin. On représente un tel graphe par une liste de nœuds `bddNode`, chacun ayant un numéro, et dans le cas d'une feuille, sa valeur (`false` ou `true`), ou dans le cas d'un nœud, le nom de la variable auquel il correspond et les numéros de ses deux sous nœuds (correspondants aux deux valeurs possibles de la variable) :

```
type bddNode =
| BddLeaf of int * bool
| BddNode of int * string * int * int;;
```

Le BDD quant à lui est la donnée de la liste des nœuds définis précédement et du numéro du nœud racine :

```
type bdd = (int * (bddNode list));;
```

Question 4 : `buildBdd : tformula -> bdd`

La fonction `buildBdd` construit le BDD d'une formule. Pour cela, il faut vérifier à chaque étape de la construction du graphe que le nœud qu'on s'apprête à construire n'est pas déjà présent dans la liste. Si ce n'est pas le cas, on l'ajoute à la liste avec un nouveau numéro. Si c'est le cas, on référence à la place le nœud déjà existant.

Par soucis de simplicité et de clareté, j'ai choisi de définir d'abord deux fonctions :

- la fonction `getBddNodeId : bddNode list -> bddNode -> int`, qui nous donne le numéro d'un nœud dans la liste s'il y est présent, et -1 sinon
- la fonction `getMaxNodeId : bddNode list -> int` qui nous renvoie le plus grand numéro attribué à un nœud dans la liste (pour pouvoir assigner un numéro à un nouveau nœud plus facilement)

Comme pour la fonction `buildDecTree`, la fonction `buildBdd` doit construire l'environnement au fur et à mesure des appels récursifs ; il est donc nécessaire, de même que pour la fonction `buildDecTree`, d'avoir un accumulateur pour l'environnement d'évaluation de la formule.

De plus la fonction `buildBdd` doit renvoyer un `bdd`, c'est-à-dire un couple `(int * bddNode list)`, j'ai donc défini une fonction auxiliaire récursive `aux`, qui prend en paramètres `bddNodeList` la liste des nœuds déjà créés, `formula` la formule, `env` l'accumulateur pour l'environnement et `vars` la liste des variables qui n'ont pas encore été ajoutées à l'environnement, et qui renvoit un `bdd` (donc un couple `(int * bddNode list)`), et non pas seulement la liste des nœuds :

```
let rec aux bddNodeList formula env vars
(* aux : bddNode list -> tformula -> env -> string
   list -> bdd *)
```

Pour déterminer si on doit créer un `BddNode` ou un `BddLeaf`, `aux` fait un pattern matching sur `vars` :

— si `vars = []`, alors tout l'environnement a été construit et donc toutes les variables se trouvent dans `env`. On crée alors un `BddLeaf` en déterminant son numéro avec `getMaxNodeId`, et on vérifie qu'il n'existe pas déjà dans le bdd avec `getBddNodeId`. Si il n'existe pas encore, on renvoit le couple `(m, node::bddNodeList)`, pour ajouter le nœud `node` nouvellement créé à la liste des nœuds (`m` étant le numéro du nouveau nœud). Sinon, on renvoit `(n, bddNodeList)`, `n` étant le numéro du nœud existant (on n'ajoute donc pas le nœud `node` à la liste).
— Si `vars` n'est pas vide (`vars = h::t`), on doit créer un `BddNode` avec les deux sous-nœuds correspondants. On fait deux appels récursifs à `aux` en récupérant la liste des nœuds mise à jour et les numéros des deux nœuds suivants, on crée notre nouveau nœud avec les numéros de ses successeurs, et on renvoit (de même que pour le cas précédent) la liste des nœuds augmentée du nœud actuel avec son numéro s'il n'est pas déjà présent dans la liste (donc `(m, node::bddNodeList)`), et le numéro du nœud déjà existant avec la liste actuelle sinon.

Il suffit ensuite d'appeler la fonction auxiliaire avec une liste de nœuds vide et un environnement vide pour obtenir le BDD complet :

```
aux [] formula [] (getVars formula);;
```

BDD simplifié

Question 5 : `simplifyBDD : bdd -> bdd`

L'objectif est maintenant de simplifier le graphe en supprimant les nœuds qui ont deux fois le même successeur. On définit pour cela la fonction `simplifyBDD : bdd -> bdd`.

Encore par soucis de simplicité, j'ai défini une fonction `simplifyNode : bddNode list -> int -> int -> bddNode list`, qui prend la liste des nœuds du graphe, deux entier `n` et `p`, et qui renvoit la liste modifiée, en ayant supprimé le nœud `BddNode(n, _, p, p)` et modifié les nœuds pointant vers `n` pour qu'ils pointent vers `p`.

La fonction `simplifyBDD` doit alors parcourir la liste des nœuds du graphe, et appeler la fonction `simplifyNode` pour chaque nœud de la forme `BddNode(_, _, p, p)`.

Bien que cela puisse paraître simple à première vue, c'est en fait plus délicat qu'on peut le penser. J'ai d'abord cherché des moyens de parcourir une liste en la modifiant en même temps, ce qui est compliqué puisqu'on n'a pas seulement besoin de modifier l'élément courant lors du parcours, mais également tous les autres nœuds qui le référencent.

La solution que j'ai trouvé a été de définir une fonction auxiliaire qui prend en argument le bdd à simplifier (qu'elle va mettre à jour à chaque

appel récursif), mais également la liste des nœuds présents initialement dans celui-ci :

```
let rec aux (bdd : bdd) (bddNodeList : bddNode list)
```

Ainsi, on peut lors des appels récursifs parcourir la liste des nœuds `bddNodeList`, en vérifiant pour chaque nœud s'il peut être supprimé du graphe, et mettre à jour le bdd complet lorsque nécessaire à l'aide de `simplifyNode`.

Il faut également penser à mettre à jour le numéro du nœud racine du BDD, si jamais celui-ci peut être simplifié.

Après avoir défini la fonction `aux` ainsi, il ne reste plus qu'à faire un pattern matching sur le bdd reçu en paramètre, pour faire l'appel initial en envoyant une "copie" de la liste des nœuds du bdd :

```
match bdd with
  | (m, l) -> aux bdd l;;
```

Vérification de formules

Question 6 : `isTautology : tformula -> bool`

Avec la simplification de la question précédente, toute tautologie se réduit à un seul nœud : la feuille true. On peut donc définir aisément une fonction `isTautology : tformula -> bool` qui vérifie si une formule est une tautologie :

```
let isTautology (formula : tformula) =
  (simplifyBDD (buildBdd formula)) = (1, [BddLeaf (1,
     true)]);;
```

Question 7 : `areEquivalent : tformula -> tformula -> bool`

On souhaite faire une fonction `areEquivalent`, qui teste si deux formules sont équivalentes. On peut simplement utiliser la fonction précédente :

```
let areEquivalent (formula1 : tformula) (formula2 :
   tformula) =
  isTautology (Equivalent(formula1, formula2));;
```

Affichage graphique de BDD

On utilise DOT pour générer graphiquement les arbres et BDD des formules. Il faut cependant générer le fichier .dot contenant la description du graphe. On définit pour cela deux fonctions :

Question 8 : `dotBDD : string -> bdd -> unit`

On créé une fonction `dotBDD : string -> bdd -> unit` qui prend un nom de fichier et un bdd et qui génère le fichier Dot correspondant.

Dans le fichier .dot, on doit identifier chaque nœud avec un numéro pour que DOT puisse faire le lien entre chaque nœud ; ce qui est plutot pratique dans le cas d'un BDD, puisque chaque nœud possède déjà un numéro.

J'ai donc défini une fonction auxiliaire récursive qui prend la liste des nœuds, et qui imprime dans le fichier les lignes correspondantes. Cette fonction fait un pattern matching sur le type du nœud, (`BddNode` ou `BddLeaf`), et imprime à l'aide de `fprintf` les lignes nécessaires pour faire apparaître dans le graphe le nœud et les deux arcs le reliant aux suivants. Par exemple,

```
fprintf file "%d [style = bold, label=\"%B\"];\n" id
    value;
```

permet d'afficher la feuille de numéro `id` et ayant comme valeur `value`. On fait ensuite un appel récursif sur la suite de la liste.

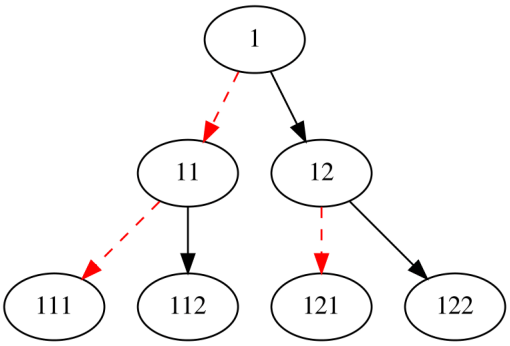
Il faut cependant imprimer au début du fichier l'instruction DOT pour commencer le graphe, et une accolade à la fin pour indiquer la fin du graphe. On utilise de même `fprintf` pour imprimer ces lignes, en appelant la fonction auxiliaire sur la liste complète entre les deux :

```
fprintf file "digraph G {\n";
match bdd with
  | (m, l) -> aux file l;
fprintf file "}";
```

Question 9 : `dotDec: string -> decTree -> unit`

Ici, on cherche à générer le fichier .dot, mais on part d'un arbre binaire. Le principe est le même, cependant cette fois ci les nœuds ne sont pas numérotés dans le `decTree` : il va falloir leur associer un numéro unique au fur et à mesure du parcours de l'arbre.

Après une certaine réflexion, j'ai eu l'idée, pour numéroter les successeurs d'un nœud de façon unique, d'ajouter respectivement 1 et 2 à la fin de la chaine de caractère du numéro du nœud parent : cela revient, pour un nœud n, à numéroter ses successeurs $10n + 1$ et $10n + 2$. On aurait ainsi la numérotation suivante :



La fonction `dotDec` est semblable à la fonction `dotBdd` précédente. On adapte seulement la fonction auxiliaire pour qu'elle prenne en plus un entier n, numéro du nœud à ajouter au fichier, et on fait les appels récursifs sur les nœuds successeurs avec respectivement $10n + 1$ et $10n + 2$ comme numéro.

Pour l'appel initial, on envoit la racine de l'arbre complet, numéroté 1 :

```
fprintf file "digraph G {\n";
aux file 1 tree;
fprintf file "}";
```

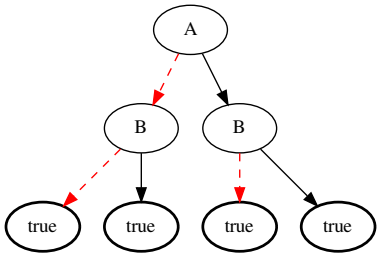
De façon plus intelligente, on aurait pu aussi remarquer qu'en assignant le numéro 1 au premier nœud et en attribuant les numéros $2n + 1$ et $2n + 2$ aux successeurs d'un nœud n, on obtient la même numérotation que si on avait attribué les numéros à chaque nœud un par un, de haut en bas et de gauche à droite.

Note

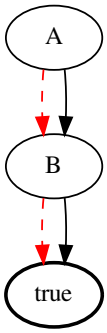
J'ai aidé certains de mes camarades pour certaines questions, en particulier pour les questions 4 et 5. Il est donc possible que quelques étudiants aient un code similaire au mien, en particulier pour ces questions.

Annexes : tests

$A \Rightarrow B \Leftrightarrow B \vee \neg A$



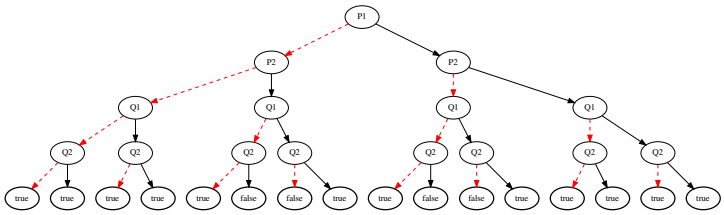
Arbre de décision



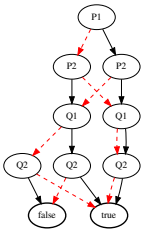
BDD

BDD simplifié

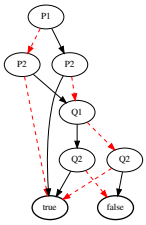
$(Q_1 \Leftrightarrow Q_2) \vee (P_1 \Leftrightarrow P_2)$



Arbre de décision



BDD



BDD simplifié