Projet reconnaisance motif désigné par une expression régulère dans un texte

Florian Devarenne MPI

Pour ce projet j'ai décidé d'implémenter l'algorithme de Berry-Sethi afin de construire un automate qui reconnait le langage désigné par notre expression régulière. Voilà le plan concret :

Etape 1 : Reconnaitre un motif dans un texte à l'aide d'un automate

Etape 2 : Construction de l'algorithme (en réfléchissant bien aux fonctions auxiliaires avant)

Etape 3 : Mise en place interface utilisation

Code disponible :
-recoreg.ml code brut avec commentaire, test etc
-recoreg_off.ml code nettoyé avec exemple laissé pour test.

Choix d'implémention

Pour représenter les automates ont choisi d'implémenter le type suivant :

```
type automate_det =
  {
    transitions : (char,int) Hashtbl.t array;
    init : int;
    final : int list;
  };;
```

parce que c'est celui qu'on a implémenté en classe (3/2) et qu'il est pratique

Etape 1 : Reconnaisance d'un motif dans un texte à l'aide d'un automate

Le problème à résoudre est le suivant :

Soit A un automate reconnaisant un langage L comment détecter si pour un texte donné T il existe des mots reconnu par L dans T.

Pour ce faire on met en place la fonction auto_reco qui marche selon l'idée suivante : faire avancer l'automate dans le texte : si il ne trouve pas de transitions on passe à la prochaine lettre si il entrouve une on continue de plus si cette etat est final on le sauvegarde dans notre liste de couple qui constitue le résultat final de la fonction.

Comme indiquez cette fonction renvoie donc une liste contenant la position et la longueur de chaque motif reconnu dans le texte. Plus exactement une liste de couple (position de la dernière lettre du motif , longueur du motif)

Pour tester et construire la fonction je me suis aider d'un automate simple reconnaisant le mot "aa". Les tests donnent :

```
# auto_reco auto_aa "bbaabbaabbaa";;
- : (int * int) list = [(2, 2); (6, 2); (10, 2)]
# auto_reco auto_aa "aa";;
- : (int * int) list = [(0, 2)]
# auto_reco auto_aa "zzaaz";;
- : (int * int) list = [(2, 2)]
```

La fonction marche parfaitement sur cette exemple. La difficulté rencontré pour la coder est qu'il est difficile de faire que l'automate s'arrête à la fin du texte : pour régler cela j'ai rajouté l'exception Invalid argument "index out of bounds" ce qui n'est pas très élégant mais cela marche et cest la seul solution que j'ai trouver. En effet je ne vois pas où je dépasse la limite des indices de mes tableaux…

J'ai aussi tester avec un exemple plus intéressant : un automate reconnaisant L tel que L = aa* . Les tests donnent :

```
#     * * * # auto_reco auto_aastar "zertazert";;
- : (int * int) list = [(4, 1)]
# auto_reco auto_aastar "bbbbaabbbb";;
- : (int * int) list = [(4, 1); (5, 2); (5, 1)]
# auto_reco auto_aastar "aaaaa";;
- : (int * int) list =
[(0, 1); (1, 2); (2, 2); (3, 2); (4, 2); (1, 1); (2, 2); (3, 2); (4, 2);
 (2, 1); (3, 2); (4, 2); (3, 1); (4, 2); (4, 1)]
#
```

Etape 2 : Construction algorithme Berry-Sethi

Maintenant que l'on possède une fonction qui reconnait un motif dans un texte à l'aide d'un automate il faut maintenant construire notre fonction permettant d'obtenir un

```
# e1;;
- : exp_reg =
Union
 (Etoile (Concat (Lettre 'a', Lettre 'b')),
  Concat (Concat (Lettre 'a', Lettre 'b'), Etoile (Lettre 'a')))
# line 0 e1;;
- : exp_reg =
Union
 (Etoile (Concat (Lettre 'A', Lettre 'B')),
  Concat (Concat (Lettre 'C', Lettre 'D'), Etoile (Lettre 'E')))
#
```

automate reconnaissant exactement le langage décrit par une expression régulière. J'ai choisi de représenter les expressions régulière de la manière suivante :

```
type exp_reg =
  |Vide
  |Eps
  |Lettre of char
  |Union of (exp_reg * exp_reg)
  |Concat of (exp_reg * exp_reg)
  |Etoile of exp_reg
;;
```

Pour arriver à notre fin ; on choisit d'implémenter l'algorithme de **BERRY-SETHI**

LINEARISATION

La première étape est de linéariser l'expression régulière. Cette étape est assez fastidieuse. J'ai fait le choix ici de renommer les lettres par les lettres de l'alphabet en majuscules successif et non par a1 pour le premier a, a2 pour le deuxième a etc car cela est trop compliqué.

La fonction finale qui linéarise notre expression régulière est :

```
let rec line (i : int) = function
```

Les différentes fonctions auxiliaire codé pour les besoins de cette fonction de linéarisation nous permettront aisément de retrouver l'expression régulière de base (Je l'ai même fait avec la fonction deline, exemple ci-dessous) :

```
# deline (line 0 e1) (list_to_tab (lettres_dans_exp_reg e1));;
- : exp_reg =
Union
 (Etoile (Concat (Lettre 'a', Lettre 'b')),
  Concat (Concat (Lettre 'a', Lettre 'b'), Etoile (Lettre 'a')))
#
```

Cela peut sembler inpertinent mais en réalité cela est utile car list_to_tab est un tableau associant chaque ancienne lettre par la lettre qui l'a remplacé dans l'expression régulière.

FIN LINEARISATION

Maintenant que l'on a linéarisé notre expression régulière on peut réflchir à comment mettre en place notre algorithme. Celui-ci va consister à construire un automate. Il faudra alors choisir l'état initial, les états finaux et créer toutes les transitions nécessaires. L'algorithme de Berry-Sethi repose sur le calcul de trois ensemble ; celui des premières lettres ; des dernières lettres et des digrammes.

Pour qu'elles soient aisément utilisable après, les fonctions auxiliares calculant ces trois ensembles renverront des listes. Je l'ai est nommé first last et digra_off. First et Last ne pose pas de problème. Par contre digra_off pose un peu plus de problème notamment pour les digrammes possibles créer par la concaténation de deux langages. Au final on y arrive bien et voici les fonctions avec un test simple sur l'expression régulière : ab*

```
# first e_simple;;
- : char list = ['a']
# last e_simple;;
- : char list = ['a'; 'b']
# digra_off e_simple;;
- : char list list = [['b'; 'b']; ['a'; 'b']]
#
```

Maintenant que l'on a ses fonctions auxilaires crucial on peut enfin coder notre fonction berry_sethi à propement parler qui a pour signature :

```
let berry_sethi (e : exp_reg) : automate_det_nonproj =
```

On remarquera que le type automate n'est plus le même, en effet cette fonction renvoie l'automate obtenu grâce à l'algo de Berry-Sethi mais pas encore projeter c'est à dire encore sur l'expression régulière linéariser. On projettera après. Le type automate_det_nonproj permet d'avoir des états marqués par des char ce qui est plus pratique dans un premier temps.

Voici un test avec l'expression régulière simple précédente ab*

d’abord sur papier :

test berry-sethi:

e_simple = ab*

↳ linéarisation

AB*

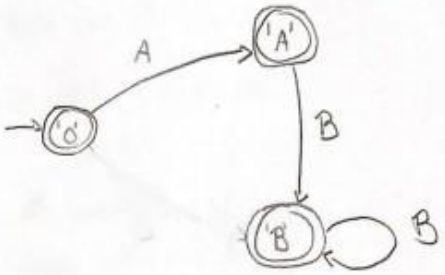
↓ calcul D, P, F

P = A (premières lettres)

D = A, B (dernières lettres)

F = AB, BB (digrammes possibles)

↓ construction de l'automate (selon algo Berry-Sethi)



Et maintenant on vérifie que c'est bien l'automate renvoyé par notre fonction Berry_Sethi:

```
# let automate_test = berry_sethi e_simple;;
val automate_test : automate_det_nonproj =
  {transitions = [|<abstr>; <abstr>; <abstr>|]; init = '0';
   final = ['A'; 'B']}
# automate_test.init;;
- : char = '0'
# automate_test.final;;
- : char list = ['A'; 'B']
# Hashtbl.find automate_test.transitions.(0) 'A';;
- : char = 'A'
# Hashtbl.find automate_test.transitions.(0) 'B';;
Exception: Not_found.
# Hashtbl.find automate_test.transitions.(1) 'A';;
Exception: Not_found.
# Hashtbl.find automate_test.transitions.(0) 'B';;
Exception: Not_found.
# Hashtbl.find automate_test.transitions.(1) 'B';;
- : char = 'B'
# Hashtbl.find automate_test.transitions.(1) 'A';;
Exception: Not_found.
# Hashtbl.find automate_test.transitions.(2) 'B';;
- : char = 'B'
# Hashtbl.find automate_test.transitions.(2) 'A';;
Exception: Not_found.
#
```

Cela correspond bien à notre automate sur notre schéma sur papier, les transitions sont bien où l'on espère et les états finaux correspond (petite erreur sur l'exemple papier , l'état initial n'est pas final). Formidable.

Le résultat est beau est satisfaisant mais le chemin pour y arriver a été semé d'embuches :
-J'ai mis longtemps à comprendre que je ne pouvais pas écrire transitions = Array.make n+1 (Hashtbl.create n+1) car cela créer la même table de hashage pour tout mon tableau et tout devenait faux Pour palier à ce problème on utilise la boucle for.
-Pour les digrammes, une liste de liste ne convient pas, il faut alors transformer les liste de deux éléments en tableaux ce qui n'est pas très élégants mais cela marche.

La dernière étape pour finir l'étape 2 est de projeter l'automate obtenu.

PROJECTION

La projection est une étape que j'avais projeté être facile mais cela n'a pas été le cas. L'idée est de procéder un peu de la même manière de berry_sethi, on remplit un automate en fonction de l'automate non projeté.

La fonction finale est de signature suivante :

```
projection (auto_bs : automate_det_nonproj) (exp : exp_reg) : automate_det =
```

On prend aussi l'expression régulière d'origine en entrée, on ne perd pas de généralité en faisant cela et c'est plus pratique dans le code.

On reprend l'exemple simple de berry sethi et on le projette, on obtient un résultat satisfaisant :

```
# let auto_simple = projection (berry_sethi e_simple) e_simple;;
val auto_simple : automate_det =
  {transitions = [|<abstr>; <abstr>; <abstr>|]; init = 0; final = [1; 2]}
# Hashtbl.find auto_simple.transitions.(0) 'a';;
- : int = 1
# Hashtbl.find auto_simple.transitions.(0) 'b';;
Exception: Not_found.
# Hashtbl.find auto_simple.transitions.(1) 'a';;
Exception: Not_found.
# Hashtbl.find auto_simple.transitions.(1) 'b';;
- : int = 2
# Hashtbl.find auto_simple.transitions.(2) 'a';;
Exception: Not_found.
# Hashtbl.find auto_simple.transitions.(2) 'b';;
- : int = 2
```

On a bien les états finaux, les transitions et l'état initial que l'on attendait, c'est bien un automate qui reconnait exactement ab*.

Magnifique !

Problème...J'ai voulu tester sur une expression régulière un peu plus compliqué ((ab)*|aba*) Tout marche bien, excepté le fait que l'automate donné par l'algorithme de berry sethi n'est pas déterministe ! Notre implémentation avec des tables de hachages ne prends pas cela en compte car elle offre la possibilité d'une seule transition par lettre partant d'un état. Pour résoudre cela il faudrait revoir l'implémentation et faire en sorte que la table de hachage associe des tableaux ou des liste d'entier à chaque caractère mais il faudrait alors changer notre fonction de reconnaisance et plein d'autre trucs donc je ne vais pas le faire ici. On pourrait aussi penser à une "déterminisation" de l'automate avant la projection.

On se contentera des expressions régulières dont l'automate de Glushkov est déterministe, c'est par exemple le cas de notre premier test ab*.

FIN PROJETION

On peut alors conclure sur cette étape 2 pour l'instant.

Etape 3 : Mise en place de l'interface d'utilisation

On prends en entrée un texte et une expression régulière donné selon notre implémentation et on renvoie la position et la longueur du motif reconnu. (cf etape 1)

On commence par des tests brut dans l'interpréteur pour vérifier que tout va bien :

```
nonproj -> exp_reg -> automate_det = <fun>
# auto_reco (projection (berry_sethi e_simple) e_simple) "abzzzzabbbbbzabzzzabb";;
- : (int * int) list =
[(0, 1); (1, 2); (6, 1); (7, 2); (8, 3); (9, 3); (10, 3); (11, 3); (13, 1);
 (14, 2); (18, 1); (19, 2); (20, 3)]
#
```

Tout ne va pas bien…Première chose positive, la position (position de la fin du motif je le rappel) de fin de chaque motif est bonne mais le problème se situe dans la longueur de ceuxci. En effet je me rends compte maintenant que la fonction auto_reco de l'étape 1 n'est pas correcte, pour compter le nombre de lettre du motif on ne peux évidemment pas se reposer sur le numéro des états, l'automate n'est pas un automate ligne…

Il y a déjà un beau résultat de connaître la position de la fin de chaque motif pour une expression régulière dont l'automate de Glushkov est déterministe.