Des automates aux expressions régulières

OPTION INFORMATIQUE - TP nº 4.3 - Olivier Reynet

```
À la fin de ce chapitre, je sais :

Coder un automate généralisé

Programmer l'élimination des états

Construire l'expression régulière au fur et à mesure de l'élimination
```

Ce TP a pour but de programmer l'algorithme d'élimination des états pour passer d'un automate à une expression régulière. Dans un premier temps, on calcule un automate correspondant à une expression régulière : cet automate est utilisé pour tester l'algorithme d'élimination des états dans la partie 3. La partie 2 permet de construire des fonctions auxiliaires utiles pour programmer l'algorithme.

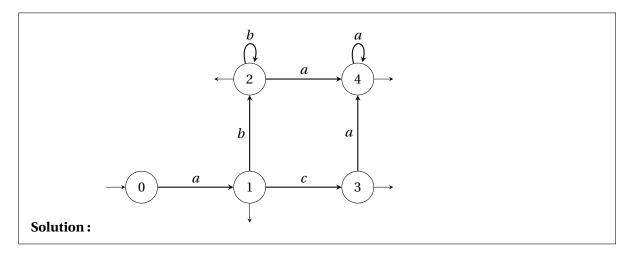
A Construction d'un automate associé à une expression régulière

On dispoes des types regexp et ndfsm suivants:

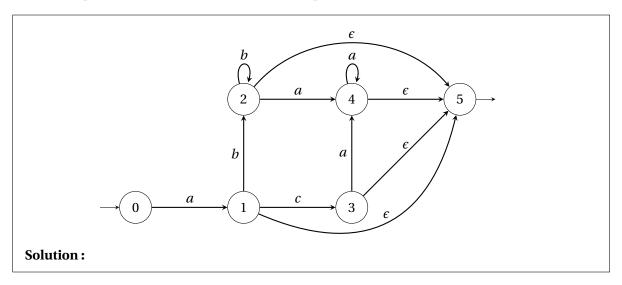
```
type regexp =
    EmptySet
    | Epsilon
    | Letter of char
    | Sum of regexp * regexp
    | Concat of regexp * regexp
    | Kleene of regexp;;

type ndfsm =
    { states : int list;
    alphabet : char list;
    initial : int list;
    transitions : (int * char * int) list;
    accepting : int list};;
```

A1. En utilisant l'algorithme de Berry-Sethi, construire l'automate de Glushkov associé à $a(b^*|c)a^*$.



A2. L'automate précédent est-il normalisé? Si ce n'est pas le cas, le normaliser.



A3. En OCaml, construire un automate de type ndfsm correspondant à $a(b^*|c)a^*$.

```
Solution:

let sigma = ['a'; 'b';'c'];;
let states = [0; 1; 2; 3; 4; 5];;
let init = [0];;
let final = [5];;
let trans = [(0,'a',1); (1,'b',2); (1,'c',3); (2,'b',2); (1,'a',4); (2,'a',4); (3,'a',4); (4,'a',4); (1,'e',5); (2,'e',5); (3,'e',5); (4,'e',5)];;

let automata = {
    states = states;
    alphabet = sigma;
    initial = init;
    transitions = trans;
    accepting = final};;
```

B Fonctions auxiliaires

Pour construire l'automate généralisé, toutes les transitions de l'automate vont devenir des expressions régulières. Par ailleurs, on cherche à traiter toutes les transitions au départ d'un même état. On s'appuie donc pour cela sur quelques fonctions intermédiaires. Pour les écrire, on pourra au choix utiliser les fonctions de la bibliothèque List ou écrire des fonctions récursives auxiliaires : il est important de savoir faire les deux.

B1. Écrire une fonction de signature label_to_regexp : char -> regexp qui transforme une lettre a de type char de l'alphabet en une expression régulière de type Letter a. Si la transition est une transition spontanée, alors on encode le mot vide par le caractère 'e'.

B2. En utilisant la fonction précédente, écrire une fonction de signature trans_to_regexp : ('a * char * 'b)list -> ('a * regexp * 'b)list qui transforme une liste de transitions labellisées par des char en une liste de transitions labellisées par des regexp.

```
Solution:
    let trans_to_regexp transitions =
        List.map (fun (q1,a,q2) -> (q1,(label_to_regexp a), q2)) transitions;;

let trans_to_regexp transitions =
        let rec map result trans =
        match trans with
        | [] -> result
        | (q1,a,q2)::t -> map ((q1,(label_to_regexp a), q2)::result) t
        in map [] transitions;;
```

B3. Écrire une fonction de signature trans_from_q: ('a * 'b * 'c)list -> 'a -> ('a * 'b * 'c)list qui extrait les transitions au départ d'un certain état q.

```
Solution:
    let trans_from_q transitions q =
        List.filter (fun (s,_,_) -> s=q) transitions;;

let trans_from_q transitions q =
        let rec filter result trans =
        match trans with
        | [] -> result
```

B4. Écrire une fonction de signature

trans_from_q_no_loop : ('a \star 'b \star 'a)list -> 'a -> ('a \star 'b \star 'a)list qui extrait les transitions au départ d'un certain état q mais pas les boucles.

```
Solution:
    let trans_from_q_no_loop transitions q =
        List.filter (fun (s,_,n) -> s = q && n != s ) transitions;;

let trans_from_q_no_loop transitions q =
        let rec filter result trans =
        match trans with
        | [] -> result
        | (s,e,n)::t -> if s = q && n != s then filter ((s,e,n)::result) t
        else filter result t
        in filter [] transitions;;
```

B5. Écrire une fonction de signature

find_loop: ('a \star 'b \star 'a)list \rightarrow ('a \star 'b \star 'a)option qui renvoie la première boucle d'une liste de transitions si une boucle existe et None sinon.

C Construire l'expression régulière

- C1. Appliquer à la main l'algorithme d'élimination des états sur l'automate normalisé de la question A.1.
- C2. Écrire une fonction de signature

```
merge_mult_trans : ('a * regexp * 'b)list -> 'a -> ('a * regexp * 'b)list qui permet de fusionner les expressions régulières associées à des transitions multiples au départ de q et à destination d'un même état. L'utilisation d'une table de hachage du module Hashtbl pour mémoriser les états suivants q déjà rencontrés et leur associer une expression régulière est recommandée. On pourra utiliser les fonction mem, find et add de ce module et transformer une Hashtbl
```

nommé dict en une liste comme suit:

Hashtbl.fold (fun key value acc -> value::acc)dict [].

- C3. Écrire une fonction de signature fsm_to_regexp : ndfsm -> regexp qui renvoie l'expression régulière associée à un automate \frac{1}{2}. On pourra procéder comme suit :
 - 1. Mettre à jour au fur et à mesure une référence vers la liste des transitions restantes.
 - 2. Supprimer les états intermédiaires qui ne sont ni accepteurs ni initial.
 - 3. S'il existe des boucles sur l'état initial ou accepteurs, les gérer puis fusionner les expressions régulières.

```
Solution:
   let fsm_to_regexp fsm =
       let trans = ref (trans_to_regexp fsm.transitions) in
       let update q =
           let qtrans = merge_mult_trans !trans q in
           let rm_state (p,e1,_) =
                   let t_loop = find_loop qtrans in
                   (* only one if one since we used merge_mult_trans *)
                   let new_t = List.fold_left
                   (fun acc (_,e2,s) ->
                   if s = q
                   then acc
                   else
                   begin
                   match t_loop with
                   | None -> (p,Concat(e1,e2),s)::acc
                     Some (_,e_loop,_) -> (p,Concat(e1,Concat(Kleene(e_loop),e2
                       )),s)::acc
                   end)
                   [] qtrans in
           trans := List.filter (fun (prev,_,next) -> not(prev = q || next = q)
               ) !trans;
           (* remove q state *)
           trans := new_t@(!trans); in (* add new transitions *)
```

^{1.} Bonne chance!