Chapitre 7 - Automates

Léo SAMUEL

30 mars 2021

1 Exemples introductifs

Ecrivons un programme pour chercher le mot "info" dans un texte. L'algo naif à déjà été vu en MPSI. Il a une complexité en $O(|\text{texte}| \cdot |\text{mot}|)$.

Si le teste échoue à la $4^{ime} lettre, on peut reprendre la recherche la 4_{ime} lettre et pas de 0$

Voici une méthode plus efficace :

— Lire une seul fois le texte et garder en mémoire où on en est du mot
Pour ce faire, utilisons le graphe suivant :
Image graphe

Mode d'emploi :
— Partir du sommet indiqué par une flèche sans sommet de départ
— lire le texte lettre après lettre et suivre les flèches
— A la fin de la lecture, si on est au sommet 4, c'est que le texte contenait "info"
L'exemple ci dessus est simple car "info" n'a pas de lettre en double. Voyons comment appliquer la méthode au mot "infini".

[Image Graphe]

Les graphes utilisé sont des "automates". L'état signalé par une flèche est dit "initial". Le ou les états signalés par 0 sont dit "acceptant" ou "terminaux"

2 Programmation d'un automate fini déterministe

On définie le type suivant :

```
type automate = { initial : int;
                  finals : int list;
                  transitions : (int*char) list array}
                ;;
```

3 Vocabulaire sur les languages

Soit Σ un ensemble qu'on appelle "alphabet"

— Un élément de Σ s'appelle une lettre
— Une suite finie de lettres s'appelle un mot
— Le mot vie est noté ϵ
— La concatenation des mots est noté $\cdot$

- On note $\Sigma*$ l'ensemble des mots sur Σ
- $\forall u \in \Sigma*, |u|$ est la longueur de u, son nombre de lettre
- On note $\forall n \in \mathbb{N}, \Sigma^n$ l'ensemble des mots de n lettres
- On identifie les mot de une lettre avec les lettres cad $\Sigma = \Sigma^1$
- Un ensemble de mot s'appelle un langage

Propriété 1 $\cdot$
- *Associatif :* $\forall u, v, w \in \Sigma, (uv) \cdot w = u(v \cdot w)$
- *Neutre :* ϵ
- *Pas Commutatif dans la plupart des cas*
- $\forall u, v \in \Sigma, |u \cdot v| = |u| + |v|$
- $\forall u, v \in \Sigma*, u = v <=> |u| = |v| \land \forall i \in [[0, |u|[[, u_i = v_i$

Définition 1 *Soit* $u, v \in \Sigma*$,

- *On dit que u est un préfixe de v lorsque* $\exists w \in \Sigma*, v = u \cdot w$
- *On dit que u est un suffixe de v lorsque* $\exists w \in \Sigma*, v = w \cdot u$

4 Définition d'un automate (fini déterministe)

Définition 2 *Un automate fini déterministe complet sur* Σ *est un quadruplet formé de*
- *Un ensemble* $\mathcal{Q}$ *appelé ensemble des états*
- *Un état paticulier* $i \in \mathcal{Q}$ *appelé l'état initial*
- *Un ensemble d'états* $\mathcal{F}\mathcal{Q}$ *appelé ensemble des états finals*
- *Une fonction* $\delta : \mathcal{Q} * \Sigma \to \mathcal{Q}$ *appelé fonction de transitions*

Remarque 1 *Dans le type Ocaml, on utiise un tableau de liste d'association pour enregistrer les transition*

Définition 3 *Soit* $(\mathcal{Q}, i, \mathcal{F}, \delta)$ *un AFDC. On définie sa fonction de transition par récurrence, ainsi :* $\delta* :$ $\mathcal{Q} * \Sigma^* \to \mathcal{Q}$ *vérifie :*

- $\forall q \in \mathcal{Q}, \delta^*(q, \epsilon) = q$
- $\forall q \in \mathcal{Q}, \forall x \in \Sigma, \forall m \in \Sigma^*, \delta^*(q, mx) = \delta(\delta^*(q, m), x)$

Propriété 2 *Soit* $(\mathcal{Q}, i, \mathcal{F}, \delta)$ *un AFDC.*
$\forall q \in \mathcal{Q}, \forall (m_1, m_2) \in \Sigma^2, \delta^*(q, m_1 m_2) = \delta^*(\delta^*(q, m_1), m_2)$

Démonstration 1 *(Récurrence sur la longueur de* m_2*)*
On fixe un $m_1 \in \Sigma^*$
$\forall n \in \mathbb{N}, P_n$ *:"*$\forall m_2 \in \Sigma^*, \delta^*(q, m_1 m_2) = \delta^*(\delta^*(q, m_1), m_2)$*"*
Initialisation : Soit $m_2 \in \Sigma^0, \delta^*(\delta^*(q, m_1), \epsilon) = \delta^*(q, m_1)$ *Hérédité : Soit* $n \in \mathbb{N}$*. Supposons* P_n*.*
Soit $m_2 \in \Sigma^{n+1}$
Soit $n_2 \in \Sigma^n, x \in \Sigma, m_2 = n_2 x$
$\delta^*(\delta^*(q, m_1), m_2)$
$= \delta^*(\delta^*(q, m_1), n_2 x)$
$= \delta(\delta^*(\delta^*(q, m_1), n_2), x)$
$= \delta(\delta^*(q, m_1 n_2), x)$
$= \delta(q, m_1 m_2 x)$

$\forall m_2 \in \Sigma^*, \delta^*(q, m_1 m_2) = \delta^*(\delta^*(q, m_1), m_2)$

Remarque 2 *On note q.m pour* $\delta^*(q, m)$ *Alors* $\forall q \in \mathcal{Q}, \forall m_1, m_2 \in \Sigma, (q.m_1).m_2 = q.(m_1 \cdot m_2)$

Définition 4 *Language reconnu par un automate*
Soit $(\mathcal{Q}, i, \mathcal{F}, \delta)$ *un AFDC que l'on note* $\mathcal{A}$*.*

- $\forall m \in \Sigma^*$*, on dit que m est reconnu par* $\mathcal{A}$ *lorsque* $\delta^*(i, m) \in \mathcal{F}$
- *On appelle langage reconnu par* $\mathcal{A}$ *l'ensemble des mots reconnu par* $\mathcal{A}$

5 Opérations sur les langages

On définit quelques opérations sur les langages On connait deja *cupetinter*, . On notera plus tard + à la place de *cup*

Définition 5 *concatenation de language Soient L_1 et L_2 deux langage. On pose $L_1L_2 = \{m_1m_2; m_1 \in L_1, m_2 \in l_2\}$*

Exemple 1 $\{ga, bu \cdot zo, meu = gazo, gameu, buzo, bumeu$

Définition 6 *Point et étoile*
- $forall langage L \forall n \in \mathbb{N}, L^n = L.L.L...L$
- $L^* = \cup(n \in \mathbb{N})L^n$

Propriété 3 *de* $\cdot$

- *associatif*
- *neutre* $\{\epsilon\}$
- *Distributif sur* $\cup$ *:* $\forall L, M, N$ *trois langages,* $L \cdot (M \cup N) = L \cdot M \cup L \cdot N$ *et* $(L \cup M) \cdot N) = M \cdot L \cup N \cdot L$

Démonstration 2 *Premiere égalité : Soit* $m \in L \cdot M \cup L \cdot N$
Traitons le cas $m \in L \cdot M$
Donc $\exists x \in L, y \in M, m = x \cdot y$ *Donc* $l \in L \cdot (M \cup N)$
Soit $m \in L \cdot (M \cup N)$
Donc $\exists x \in L, \exists y \in M \cup N, m = x \cdot y$ *alors* $m \in L \cdot M \cup L \cdot N$

Remarque 3 *Dans ce chapitre, on note parfois + au lieu de* $\cup$*. De plus, le neutre pour + sera* $\emptyset$

Propriété 4
- $\forall$ *langages* $L, (L^*)^* = L^*$
- $\forall L, M \in \mathcal{P}(\Sigma^*), L \subset M \Rightarrow L^* \subset M^*$*. On dit que* $*$ *est croissante*

Démonstration 3 *Supposons* $L \subset U$
Soit $m \in L^*$*,* $\exists p \in \mathbb{N}, (l_1, \ldots, l_p) \in L^p$ *tq* $m = l_1l_2 \ldots l_p$

Or $\forall i \in [[1, p]], l_i \in L \subset M$ *Donc* $m \in M^*$

Définition 7 *Langage régulier*
Un langage est dit régulier lorsqu'il peut être décrit à l'aide d'un nombre fini d'opérations :
- + *ou* $\cup$
- $\cdot$
- $*$
- $\emptyset$
- ϵ
- *Les singletons* $\{x\}$ *pour* $x \in \Sigma$

Une telle formule est appelé une expression régulière (cf suite du cours)

Remarque 4 *Il n'y à pas de* $\cap$ *ni de complémentaire dans cette définition*

Exemple 2
- *Ensemble des texte contenant le mot info :* $\Sigma^* \cdot info \cdot \Sigma^*$
- *Soit* $\Sigma = \{a, b\}$*. Ensemble des mots contenant deux a :* $b^*ab^*ab^*$

Remarque 5 $\forall x \in \Sigma$*, on notera* x *au lieu de* $\{x\}$

6 Automate incomplet

6.1 Définition

La définition d'un AFD incomplet est la même que celle d'un AFD complet sauf que la fonction de transition δ n'est pas définie sur $\mathcal{Q} * \Sigma$

Un couple $(q, x) \in \mathcal{Q} * \Sigma$ où δ n'est pas définie s'appelle un blocage

La fonction de transition étendue δ^* est alors défine seulement sur une partie de $\mathcal{Q} * \Sigma$
Un mot $m \in \Sigma^*$ est reconnu lorsque $(i, m) \in D_{\delta^*}$ $\delta * (i, m) \in \mathcal{F}$

AFD non complet pour reconnaitre l'ensemble des numéros de téléphone

- Si on lit autre chose qu'un chiffre, c'est un blocage
- Si on lit plus de 10 chiffre, c'est encore un blocage

Un automate incomplet est alors

- Plus pratique à dessiner
- Plus clair
- Moins gourmant en mémoire et en temps : Les listes d'association sont plus courte et le programme peut s'arreter avant la fin

6.2 Programmation

```
exception Blocage;;

let delta2 i x a=
  (*  a : automate
      x : une lettre
      i : un état de a
  *)
  let rec aux = function
    | [] -> raise Blocage
    | (lettre, etat)::_ when lettre = x -> etat
    | _::q -> aux q
    in
    aux (a.transitions.(i))
;;

let delta_etoile2 i m a =
  (* Cette fois m est une chaine de caracteres.
      Renvoie l'etat atteint apres lecture de toutes les lettres de m *)

  let rec boucle q k =
      (* k : prochaine lettre de m à lire
          q : etat actuel *)
          if k = String.length m then
              q
          else
              boucle (delta2 q m.[k] a ) (k+1)
  in
  boucle i 0
;;

let reconnu2 m a =
  try
    List.mem (delta_etoile2 a.initial m a) a.finals
  with
    | Blocage -> false
```

```
;;

let completed a alphabet=
  let n = Array.length a.transitions in
  let p = n in
  let nv_trans = Array.make (p+1) (tout_vers p alphabet) in
  for i=0 to n-1 do
    nv_trans.(i) <- (a.transitions.(i)@ nv_trans.(i));
  done;
  {
    initial = a.initial;
    finals = a.finals;
    transitions=nv_trans
  }
;;
```

6.3 Complétion

Dans certain cas, on a besoin de completer un automate incomplet.

Théorème 1 *Soit $(\mathcal{Q}, i, \mathcal{F}, \delta)$ un AFD qu'on note $\mathcal{A}$*
On définit un nouvel automate $\mathcal{A}'$ ainsi
— *Soit p un nouvel état et $\mathcal{Q}' = \mathcal{Q} \cup \{p\}$*
— *On garde i et $\mathcal{F}$*
— *On garde les transitions de $\mathcal{A}$*
— *On rajoute $\forall (q, x) \in \mathcal{Q} * \Sigma$ blocage dans $\mathcal{A}$ q->p et p->p*
Alors $\mathcal{A}'$ est complet et il reconnait le même langage que $\mathcal{A}$

Démonstration 4 *Soit $m \in \mathcal{L}(\mathcal{A})$. Soit c le chemin de $\mathcal{A}$ étiqueté par m. Ce chemin existe aussi dans $\mathcal{A}'$. Donc m est reconnu par $\mathcal{A}'$*
[$\Rightarrow$] Soit $m \in \mathcal{L}(\mathcal{A}')$Soit c un chemin dans $\mathcal{A}'$ étiqueté par m, de i à un certain q_f.

c ne passe pas par p sans quoi il finirait à p or $q_f \neq p$ car $p \notin \mathcal{F}$
Donc toutes les transitions emprunté par c existent dans $\mathcal{A}$
Donc c est un chemin de $\mathcal{A}$

Corolaire 1 *L'ensemble des language reconnaissable par un AFD complet est le même que l'ensemble des language reconnaissable par un AFD pas forcément complet*

7 Automate émondé

But : "alleger" un automate en suprimant des états inutiles

7.1 Définitions

Soit $(\mathcal{Q}, i, \mathcal{F}, \delta)$ un AFD qu'on note $\mathcal{A}$

— $\forall q \in \mathcal{Q}, q$ est dit accessible lorsque $\exists m \in \Sigma^*$ tq $\delta^*(i, m) = q$
— $\forall q \in \mathcal{Q}, q$ est dit co-accessible lorsque $\exists m \in \Sigma^*$ tq $\delta^*(i, m) \in \mathcal{F}$
— $\mathcal{A}$ est dit "émondé" lorsque tous ses états sont accessible et co-accessible
On pourrait qualifier d'inutile tout état non accessible ou non co-accessible

7.2 Emondage

Soit $(\mathcal{Q}, i, \mathcal{F}, \delta)$ un AFD qu'on note $\mathcal{A}$
Soit $\mathcal{Q}$ l'ensemble des états accessibles et co-accessibles
Soit la restriction de δ à $\mathcal{Q}$

Soit $(\mathcal{Q}_{,i,\mathcal{F}\cap\mathcal{Q},\delta_{)}}$ un AFD qu'on note $\mathcal{A}$
Alors $\mathcal{A}$ est émondé et $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{)}$
Ainsi quand on a obtenu un AFD qui reconnait un certain langage, on a presque toujours intérêt à l'émonder pour obtenir un automate plus simple reconnaissant le même langage.

Démonstration 5 - *Montrons que $\mathcal{A}$ est émondé. Cad montrons que les états de $\mathcal{A}$ sont accessibles et co-accessible dans $\mathcal{A}$ (sachant qu'ils le sont dans $\mathcal{A}$)*
Soit $q \in \mathcal{Q}$. Donc q est accessible et co-accessible
Donc $\exists\gamma$ chemin dans $\mathcal{A}$ qui part de i et arrive à un état de $\mathcal{F}$.
Tous les états le long de γ sont accessibles et co-accessibles donc sont encore dans $\mathcal{Q}$ Donc $\gamma \subset \mathcal{Q}$. Donc q est accessible et co-accessible dans $\mathcal{A}$ Donc $\mathcal{A}$ est émondé
- Montrons que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{)}$
Soit $m \in= \mathcal{L}(\mathcal{A}_{)}$ Soit γ un chemin acceptant dans $\mathcal{A}$ étiqueté par m. Alors γ est aussi un chemin acceptant dans $\mathcal{A}$
Donc $m \in \mathcal{L}(\mathcal{A})$

Soit $m \in= \mathcal{L}(\mathcal{A})$. Soit γ un chemin acceptant dans $\mathcal{A}$ étiqueté par m
Avec le même raisonnement, $m \in \mathcal{L}(\mathcal{A}_{)}$

7.3 Programmation

- Trouver les sommets accessibles $\rightarrow$ il suffit delancer un parcours de graphe depuis le sommet initial
- Trouver les sommets co-accessibles
  - Créer un tableau coaccessible, initialement rempli de false
  - rajouter en argument le chemin parcouru pour arriver au sommet actuel

---

```
let rec accessible trans i=
        let n = Array.length trans in
        let deja_vu = Array.make n false in
        let rec visite_sommet s=
                if deja_vu.(s) then []
                else (
                        deja_vu.(s) <- true;
                        s::visite_voisins trans.(s)
                )
        and visite_voisins = function
                | [] -> []
                | (_,q)::suite -> (visite_sommet q) @ (visite_voisins suite)
        in
                visite_sommet i
;;

let etats_utiles a =
        let trans = a.transitions in
        let n = Array.length trans in
        let deja_vu = Array.make n false in
        let coaccessible = Array.make n false in

                let rec visite_sommet s chemin_parcouru=
                        if deja_vu.(s) then []
                        else (
```

```
                                deja_vu.(s) <- true;
                                if List.mem s (a.finals) then
                                        List.iter (fun q -> coaccessible.(q)<- true)
                                                                                (s::chemin_parcouru);
                                s::visite_voisins (s::chemin_parcouru) trans.(s)
                        )
                and visite_voisins chemin_parcouru = function
                        | [] -> []
                        | (_,q)::suite -> (visite_sommet q chemin_parcouru) @ (visite_voisins chemin_parcouru suite)
                in
                        List.filter (fun q -> coaccessible.(q))
                                                                        (visite_sommet a.initial [])
;;


let emonde a=
        let liste_etats_utiles = etats_utiles a in
        let n = Array.length a.transitions in

        let sans_transition_inutile l=
                List.filter (fun (_,q) -> List.mem q liste_etats_utiles) l
        in

        for i=0 to n-1 do
                if not (List.mem i liste_etats_utiles) then
                        a.transitions.(i) <- []
                else
                        a.transitions.(i) <- sans_transition_inutile a.transitions.(i)
        done;
;;
```

8 Automate non déterministe

8.1 Principe

Pour un état q et une lettre x, on autorise plusieurs transition depuis q étiquetées par x. Donc pour un même mot m, il peut y avoir plusieurs chemin étiquetés par m. m sera reconnu si au moins un de ces chemin arrive dans $\mathcal{F}$

Ces automates sont beaucoup plus simple à concevoir et à créer que les AFD vus au début du cours. Le défaut est qu'il est plus compliqué à programmer et plus lent

En effet à chaque lettre on doit calculer δ pour chaque élement d'une liste d'états au lieu d'un seul état. Il est possible de déterminiser automatiquement n'importe quel AFND

8.2 Définition précise

Un automate non deterministe sur Σ est un quadruplet formé de
— Un ensemble $\mathcal{Q}$ appelé ensemble des états
— Une partie $\mathcal{I}$ de $\mathcal{Q}$ appelé ensemble des états initiaux
— Une partie de $\mathcal{Q}$ appelé ensemble des états finals
— Une fonction $\delta : \mathcal{Q} \times \Sigma \longrightarrow \mathcal{P}(\mathcal{Q})$ appelée fonction de transition
Pour $q \in \mathcal{Q}, x \in \Sigma, \delta(q,x)$ peut être $\emptyset$. C'est l'analogue d'un blocage
Soit $(\mathcal{Q}, \mathcal{I}, \mathcal{F}, \delta)$ un AFND. On défini la fonction de transition étendue δ^* par récursivité ainsi :
— $\forall q \in \mathcal{Q}, \delta^*(q, \epsilon) = \{q\}$
— $\forall q \in \mathcal{Q}, \forall m \in \Sigma^*, \forall x \in \Sigma, \delta^*(q, mx) = \cup_{r \in \delta^*(q,m)} \delta(r, x)$

Soit A un AFDN et $(\mathcal{Q}, \mathcal{I}, \mathcal{F}, \delta)$ ses composantes.
$\forall m \in \Sigma^*$, on dit que m est reconnu par $\mathcal{A}$ lorsque $\exists i \in \mathcal{I}, \exists f \in \mathcal{F}$ tq $f \in \delta^*(i, m)$ ou $\cup_{i \in \mathcal{I}} \delta^*(i, m) \cap \mathcal{F} \neq \emptyset$

On note $\mathcal{L}(\mathcal{A})$ l'ensemble des mots reconnus par $\mathcal{A}$

Interpretation en terme de chemin : $\forall m \in \Sigma^*, m \in \mathcal{L}(\mathcal{A}) \Leftrightarrow$ il existe un chemin étiqueté par m partant d'un état initial arrivant à un état final

8.3 Programmation

INSERT CODE TODO

8.4 Determinisation

Soit $(\mathcal{Q}, \mathcal{I}, \mathcal{F}, \delta)$ un AFND noté $\mathcal{A}$. On définit l'AFD $\mathcal{A}_d$ ainsi :

— On pose $\mathcal{Q}_d = \mathcal{P}(\mathcal{Q})$. Ainsi un état de $\mathcal{A}_d$ est un ensemble d'états de $\mathcal{A}$. On appellera $\mathcal{A}_d$ l'automate des parties de $\mathcal{A}$

— On prend comme état initial $\mathcal{I}$

— On pose $\delta_d : \substack{\mathcal{Q}_d \times \Sigma \longrightarrow \mathcal{Q}_d \\ (X,x) \to \bigcup_{q \in X} \delta(q,x)}$

— On pose $\mathcal{F}_d = \{X \in \mathcal{P}(\mathcal{Q}) | X \cap F \neq \emptyset\}$

On pose alors $\mathcal{A}_d = (\mathcal{Q}_d, \mathcal{I}, \mathcal{F}_d, \delta_d)$. C'est un AFD

Principe : $\forall X \in \mathcal{P}(\mathcal{Q}), \forall m \in \Sigma^*, \delta_d^*(X, m)$ est l'ensemble des états accessibles en lisant m depuis un élément de X.

$\forall X \in \mathcal{P}(\mathcal{Q}), \forall m \in \Sigma^*, \delta_d^*(X, m) = \bigcup_{r \in X} (r, m)$

Démonstration 6 *$\forall n \in \mathbb{N}$, soit $\mathcal{P}(n)$: "$\forall m \in \Sigma^*, \forall X \in \mathcal{P}(\mathcal{Q}), \delta_d^*(X, m) = \bigcup_{r \in X} (r, m)$"*
IT
: Soit $m \in \Sigma^0$ alors $m = \epsilon$. Alors $\delta_d^*(X, \epsilon) = X$ et $\bigcup_{r \in X} (r, \epsilon) = \bigcup_{r \in X} \{r\} = X$. Donc $\mathcal{P}(0)$ [HERE] : Soit $n \in \mathbb{N} tq \mathcal{P}(n)$. Soit $m \in \Sigma^{n+1}$ et $X \in \mathcal{P}(\mathcal{Q})$. Soit $m' \in \Sigma^n$ et $x \in \Sigma tq m = m'x$ On a $\delta_d^*(X, m) = \delta_d^*(X, m'x) = \delta_d(\delta_d^*(X, m'), x) = \bigcup_{q \in \bigcup_{r \in X} \delta^*(r, m')} \delta(q, x) = \bigcup_{r \in X} \bigcup_{q \in \delta^*(r, m')} \delta(q, x) = \bigcup_{r \in X} \delta^*(r, m'x)$

$\mathcal{L}(\mathcal{A}_d) = \mathcal{L}(\mathcal{A})$

Un langage est reconnaissable par un AFND ssi il est reconnaissable par un AFD

Démonstration 7 *Soit $m \in \Sigma^*$. $m \in \mathcal{L}(\mathcal{A}) \Leftrightarrow \exists q \in \mathcal{I} tq \delta^*(q, m) \cap \mathcal{F} \neq \emptyset \Leftrightarrow \bigcup_{q \in \mathcal{I}} \delta^*(q, m) \cap \mathcal{F} \neq \emptyset \Leftrightarrow \delta^*(q, m) \in \mathcal{F}$*

Soit l'automate exemple de la partie précedente qui reconnait "ici".
Calculons $\mathcal{A}_d$. On prend $\mathcal{Q}_d = \mathcal{P}([[0, 3]])$ On va dessiner uniquement les états accessibles

— état initial : $\{0\}$

— états finals : tous les ensembles qui contiennent 3 (il y en a 8)

— On va faire un tableau et le remplir ligne après ligne

METTRE LE TABLEAU TODO

8.5 Programmation de la determination

On a déjà δ_d. La seule difficulté, en Ocaml, un état est representé par un entier. Il faut attribuer un entier à chaque élément de $\mathcal{Q}_d$ On va numéroter uniquement les états accessible par le même algo que l'exemple precédent

Etape 1 : numéroter les états accessibles dans $\mathcal{A}_d$. Les états seront représentées par des int list strictements croissantes. On va de plus numéroter ces états grace à des dictionnaire (int list -> int). On enregistrera aussi un tableau pour associer à chaque numéro l'état correspondant.

9 Expressions régulières

On définit les expressions regulieres sur Σ :

— $\emptyset \epsilon$ et les lettres sont des regex

— Si e est une regex, alors (e^x) aussi

— Si e, f sont deux regex, alors $(e+f)$ et $(e \cdot f)$ aussi

On définit le type Ocaml par :

TODO mettre le type ocaml

Notons $\mathcal{R}(\Sigma)$ l'ensemble des regex sur Σ On définit la fonction $\mathcal{L} : \mathcal{R}(\Sigma) \to \mathcal{P}(\Sigma^*)$ par recursivité :

— $\mathcal{L}(\emptyset) = \emptyset$
— $\mathcal{L}(\epsilon) = \{\epsilon\}$
— $\forall x \in \Sigma, \mathcal{L}(x) = \{x\}$
— $\forall e \in \mathcal{R}(\Sigma), \mathcal{L}(e*) = \mathcal{L}(e)*$
— $\forall (e,f) \in \mathcal{R}(\Sigma), \mathcal{L}(e+f) = \mathcal{L}(e) \cup \mathcal{L}(f)$
— $\forall (e,f) \in \mathcal{R}(\Sigma), \mathcal{L}(e \cdot f) = \mathcal{L}(e) \cdot \mathcal{L}(f)$

Cette propriété permet de se passer des constructeurs $\emptyset$ et ϵ

Soit e une regexp ni $\emptyset$ ni ϵ, alors il existe e' une regexp ne contenant ni $\emptyset$ ni ϵ tq $\mathcal{L}(e) = \mathcal{L}(e')$

Par induction structurelle :

— Soit e un lettre : e ne contient ni $\emptyset$ ni ϵ alors $e' = e$ convient
— Soit $f \in \mathcal{R}(\Sigma)$ et $e = f$. Supposons la prop vraie pour f. Soit f' une regexp sans $\emptyset$ ni ϵ. Alors $\mathcal{L}(e) = \mathcal{L}(f^*) = \mathcal{L}(f)^* = \mathcal{L}(f)^* ou(\mathcal{L}(f') \cup \{\epsilon\}) = \mathcal{L}(f')^*$.
  Alors on prend $e' = f'*$ TODO ajouter la fin de la demo

En consequence de cette proposition, on considerera dans la suite des regexp construits sans ϵ ni $\emptyset$.

10 Langages Locaux

10.1 Programmation

TODO AJOUTER CODE

10.2 Automate associé à un langage

Si $\mathcal{L}(e)$ n'est pas local, on peut quand même calculer les ensembles. Dans ce cas on aura $\mathcal{L}(e) \neq \mathcal{L}(\mathcal{P}, \mathcal{S}, \mathcal{F}, \alpha)$

Soit $\mathcal{L}$ un langage local et $(\mathcal{P}, \mathcal{S}, \mathcal{F}, \alpha)$ ses paramètres

On créer un état par lettre et un état qui servira d'état initial.

$\forall x \in \Sigma$, notons q_x l'état associé à x et notons q_0 l'état supplémentaire.

Soit $\mathcal{Q} = \{q_0\} \cup \{q_x ; x \in \Sigma\}$

- Etat initial : q_0

- Transitions :

— $\forall a, b \in \mathcal{F}$, on ajoute la trans $q_a \to b \to q_b$
— $\forall x \in \mathcal{P}$, on ajoute la trans $q_0 \to x \to q_x$

- Etats finals :

— On prend q_0 ssi α
— Et on prend les $\{q_s ; s \in \mathcal{S}\}$

Soit $e = a^*(b+c)^*d$ On a $\mathcal{P} = \{a,b,c,d\}$ $\mathcal{S} = \{d\}$ $\alpha = \perp$ $\mathcal{F} = \{aa, ab, ac, bb, cc, bc, cb, bd, cd, ad\}$

On consate que cet automate reconnait $\mathcal{L}(e)$. De plus, il est local

Si $\mathcal{L}$ est local, $\mathcal{A}$ reconnait $\mathcal{L}$

Soit $m \in \mathcal{L}$

— Si $m = \epsilon$ alors α donc q_0 est final. Or $\delta^*(q_0, epsilon) = q_0$ donc $\epsilon \in \mathcal{L}$
— Si $m \neq \epsilon$, on a $m = m_0 ... m_{n-1}$.
  — $m_0 \in \mathcal{P}$ donc on a la transition $q_0 \to m_0 \to q_{m_0}$
  — $\forall i \in [[0, n-1[[, m_i m_{i+1} \in \mathcal{F}$ [1]

1. Ce n'est pas l'ensemble des états finaux

Ainsi, $\mathcal{A}$ contient le chemin.
Donc $m \in \mathcal{L}(\mathcal{A})$

Soit $m \in \mathcal{L}(\mathcal{A})$

- Si $m = \epsilon$, q_0 est final, donc $\alpha = \top$ donc $\epsilon \in \mathcal{L}$
- Sinon : $m = m_0 ... m_{n-1}$
  - Pas de blocage en lisant m_0 donc il existe une transtion $q_0 \to m_0 \to q_{m_0}$ Donc $m_0 \in \mathcal{P}$
  - $\forall i \in [[0, n-1[[$, apres lecture de m on est dans q_{m_i} et il n'y a pas de blocage ensuite. Alors la transition $q_{m_1} \to m_i \to q_{m_{i+1}}$
- apres lecture de m on est dans $q_{m_{n-1}}$ Or cet état est final car m est reconnu

Ainsi $m \in \mathcal{L}(\mathcal{P}, \mathcal{S}, \mathcal{F}, \alpha)$

L'interêt des