Automates finis déterministes

OPTION INFORMATIQUE - TP n° 3.9 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ définir un automate fini déterministe
- ☞ représenter un automate fini déterministe
- ☞ qualifier les états d'un automates (accessibilité)
- ☞ compléter un AFD
- ☞ complémenter un AFD
- ☞ faire le produit de deux AFD

A Construction d'automates simples

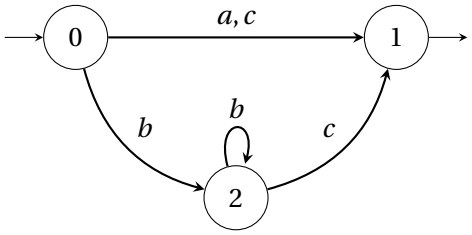
A1. On considère l'alphabet $\Sigma = \{a, b, c\}$. Construire les automates suivants :
   (a) $\mathcal{A}_0$ reconnaissant les mots commençant par deux occurrences de a.
   (b) $\mathcal{A}_1$ reconnaissant le langage défini par l'expression rationnelle $a|b^* c$.
   (c) $\mathcal{A}_2$ reconnaissant les mots contenant un nombre de a égal à 1 modulo 3, sans contrainte sur les autres lettres.
   (d) $\mathcal{A}_3$ reconnaissant les mots contenant un nombre pair de a et un nombre impair de b, c'est à dire $\{b, baa, aab, bbb, bababbb, aabbb, ababaab, \ldots\}$. On suppose que l'alphabet est $\Sigma = \{a, b\}$.

A2. Donner les représentations tabulaires des automates $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$.

A3. Les automates que vous avez dessinés sont-ils complets ?

A4. Combien d'automates complets différents à n états peut-on construire ? On cherchera à exprimer la réponse en fonction de n et $|\Sigma|$

B Complété et complémentaire d'automates finis déterministes



B1. Compléter l'automate fini déterministe $\mathcal{A}$ suivant.

B2. Quels sont les états co-accessibles de l'automate complété de $\mathcal{A}$?

B3. Dessiner le complémentaire de l'automate $\mathcal{A}$ précédent.

B4. Le complémentaire de l'automate $\mathcal{A}$ reconnaît-il le mot vide ϵ ?

C Modélisation d'un automate en OCaml

On choisit de modéliser un automate fini par un type algébrique de la manière suivante :

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

Les états sont représentés par des types `int`. Les lettres sont des types `char`. On précise l'état initial (unique) et les états accepteurs par une liste. La fonction de transition est une liste de triplets matérialisant chaque transition sous la forme `(q,'a',q')` s'il existe une transition menant de `q` à `q'` lorsqu'on a reçu la lettre `a`.

C1. Créer une variable `automata` qui représente l'automate $\mathcal{A}$ de la section B.

C2. Créer une fonction récursive de signature
`next_state : 'a -> 'b -> ('a * 'b * 'c)list -> 'c option`
et qui calcule l'état suivant, étant donné un état courant, une lettre et la liste des transitions. Le type renvoyé est optionnel, c'est à dire qu'il se peut qu'il n'y ait pas d'état suivant. Dans ce cas, la fonction renvoie `None`.

C3. Quelle est la complexité dans le pire et le meilleur des cas de la fonction précédente ? Pourrait faire mieux dans le pire des cas avec une autre structure de données ?

D Calcul d'un mot par un automate

D1. Créer une fonction de signature `up_to : fsm -> string -> int` qui renvoie l'état d'arrivée dans lequel se trouve un automate à qui on a donné un mot reconnaître.

D2. Créer une fonction de signature `match_word : fsm -> string -> bool` qui renvoie vrai si un mot est reconnu par l'automate, faux sinon. Se servir de la fonction précédente et de la fonction `List.mem`.

E Algorithmes de transformation simple d'un automate

E1. Créer une fonction de signature `get_alphabet : fsm -> char list` qui renvoie l'alphabet utilisé par l'automate. L'alphabet obtenu est-il toujours complet ?

E2. Créer une fonction de signature `succ : fsm -> int -> int list` qui renvoie la liste des successeurs d'un état donné de l'automate. On admet qu'un même état puisse être présent plusieurs fois dans la liste.

E3. Créer une fonction de signature `prev : fsm -> int -> int list` qui renvoie la liste des prédécesseurs d'un état donné dans un automate. On admet qu'un même état puisse être présent plusieurs fois dans la liste.

E4. Créer une fonction de signature qui teste la complétude d'un automate.

complétion : tester la complétude puis algo de complétion p106 107 complémentaire : facile p108

F États accessibles et co-accessibles d'un automate

p95 falcone p96
Produire l'algo d'un automate émondé p111

G Problème de la finitude d'un langage reconnaissable

p100