TP : Automates finis : suite

Nous reprenons le TP précédent sur les automates là où il a été arrêté. J'ai ajouté une partie sur les automates non déterministes.

Rappels : Nous typons les automates de manière similaire au cours :

```
type etat = int;;

type alphabet = char list;;

type auto = {
    taille: int;
    init: etat;
    final: etat list;
    trans: (char * etat) list array};;
```

Ce type permet de représenter à la fois les automates finis déterministes et non déterministes.

On a déjà codé les fonctions

- `lecture_afd : auto -> etat -> string -> etat`
- `recon_afd : auto -> string -> bool`

1 Quelques constructions classiques

1.1 Complétion d'un automate

Soit A un automate fini déterministe on veut construire un afd A' reconnaissant le même langage. On procédera comme en cours en ajoutant un état puits à l'automate.

1. Réécrire la fonction `map : ('a -> 'b) -> 'a list > 'b list` telle que `map f l` calcule la liste des images des éléments de `l` par `f`.
2. Écrire une fonction `ajoute_fleches : alphabet -> etat -> ((char * etat) list) -> ((char * etat) list)` qui étant donné un alphabet et un état puits, ajoute à une liste de flèches les flèches étiquetées par les lettres manquantes en les faisant pointer vers l'état puits.
3. En déduire une fonction `completion : alphabet -> auto -> auto` qui complète un afd étant donné un alphabet.
4. Quelle est la complexité dans le pire cas de la fonction `completion` en fonction de $|\Sigma|$ et de la taille de l'automate ?
5. Compléter l'automate A_1.

1.2 Complémentaire d'un automate

Soit A un afd. On rappelle que pour construire un automate reconnaissant $\Sigma^* - L(A)$ il suffit de compléter A puis d'inverser les états finaux et non finaux.

1. Écrire une fonction `comp_int : int -> int list -> int list` telle que `comp_int n l` calcule une liste des entiers de $[|0, n|]$ n'appartenant pas à l.
2. Écrire la fonction `comp : alphabet -> auto -> auto` calculant un automate reconnaissant le complémentaire d'un langage reconnu par un autre automate.
3. Construire avec ce qui précède un afd reconnaissant les mots de Σ^* ne commençant pas par ab.

1.3 Automate produit

On s'intéresse maintenant à l'automate produit de deux afd A_1 et A_2. Pour rappel, l'automate produit consiste à exécuter simultanément A_1 et A_2. Voici sa définition : $A = A_1 \times A_2 = (Q, q_0, F, \delta)$ est défini par

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- $F = F_1 \times F_2$
- $\forall a \in \Sigma,\ \delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ sous réserve d'existence

Si $|Q_1| = n_1$ et $|Q_2| = n_2$ l'automate produit a donc $n_1 n_2$ états. On aura donc intérêt à considérer une bijection

$$\varphi : [|0, n_1 - 1|] \times [|0, n_2 - 1|] \to [|0, n_1 n_2 - 1|].$$

1. Écrire une fonction `bijection : int -> int -> (((int * int) -> int) * (int -> (int * int))` tel que `bijection n1 n2` renvoie un couple `(phi, psi)` contenant la bijection d'intérêt φ et sa réciproque ψ.
2. Écrire une fonction `produit_cartesien : 'a list -> 'b list -> ('a * 'b) list` telle que `produit_cartesien l1 l2` construit la liste des couples (x_1, x_2) avec $x_1 \in l_1$ et $x_2 \in l_2$. On ne cherchera pas à optimiser la complexité et on pourra utiliser la concaténation de listes.
3. Écrire une fonction `produit_fleches : (int -> (int * int)) -> (char * etat) list -> (char * etat) list -> (char * etat) list` telle que `produit_fleches phi l1 l2` construit la liste associative correspondant aux transitions sortantes de l'état $\varphi(q_1, q_2)$ où `l1` est la liste associative des transitions sortantes de q_1 et `l2` la liste associative des transitions sortantes de q_2.
4. Écrire la fonction `produit : auto -> auto -> auto` réalisant le produit d'automates.
5. En déduire un automate reconnaissant les mots ne commençant pas par ab et contenant un nombre pair de a
6. En déduire de même un automate reconnaissant les mots u tels que $|u|_b \equiv 2 \pmod 3$ et contenant un nombre pair de a.

2 Automates non déterministes

On choisit de représenter un automate fini non déterministe par le type suivant proche de la définition vue en cours :

```
type afnd = {
    taille: int;
    init: etat list;
    final: etat list;
    trans: etat -> char -> etat list};;
```

On supposera que les valeurs de type `etat list` sont des listes triées par ordre strictement croissant.

Construire et coder un automate non déterministe reconnaissant les mots finissant par *bab*.

2.1 Représentation des parties de $\mathbb{N}$ à l'aide de listes triées

Il nous sera utile de pouvoir manipuler des parties de $\mathbb{N}$ à l'aide de listes triées sans répétition. Programmer en Caml les manipulations ensemblistes suivantes :

1. `union : int list -> int list -> int list`
2. `inter : int list -> int list -> int list`
3. `estvide : int list -> bool`
4. Quel est le coût de ces opérations ?

2.2 Calcul d'un afnd

1. Pour simplifier l'écriture des programmes on transformera les chaînes de caractères en listes de caractères. Écrire la fonction `liste_lettres : string -> char list` correspondante.
2. Écrire la fonction de transition étendue aux mots `delta : afnd -> etat -> char list -> etat list`
3. En déduire la fonction `recon_afnd : afnd -> char list -> bool` testant la reconnaissance d'un mot par un automate non déterministe.

3 Déterminisation d'un automate

Dans cette section on se propose de déterminiser un afnd comportant un maximum de 30 états.

La stratégie consiste à représenter une partie de $X \subset [|0, 29|]$ par l'entier

$$n_X = \sum_{i=0}^{29} 2^i \times 1_X(i)$$

où 1_X est la fonction indicatrice de la partie X.

On ne remplira la table de transition et la liste des états finaux que pour les états accessibles de l'automate des parties.

1. Écrire une fonction `codage_liste : int list -> int` prenant une liste d'entiers triée représentant une partie X et retournant n_X.
2. Écrire la fonction réciproque `decodage_entier : int -> int list`.
3. Écrire des fonctions de manipulation d'ensembles à l'aide de leur codage, on pourra utiliser astucieusement les opérations processeurs bit à bit
   (a) L'union : calculer $n_{X\cup Y}$ à partir de n_X et n_Y
   (b) L'intersection : calculer $n_{X\cap Y}$ à partir de n_X et n_Y
   (c) Le test d'appartenance : tester si $x \in X$ à partir de n_X
   (d) Le test du vide : tester si $X = \varnothing$ à partir de n_X
4. Écrire une fonction `etat_initial : afnd -> int` retournant l'état initial de l'automate des parties.
5. Écrire une fonction `est_final : afnd -> int -> bool` testant si n_X est un état final de l'automate des parties
6. Écrire une fonction `delta : afnd -> alphabet -> int -> char -> int` codant la fonction de transition $\delta : \mathcal{P}(X) \times \Sigma \to \mathcal{P}(X)$ de l'automate des parties.
7. (Difficile) En déduire une fonction `determinisation : afnd -> alphabet -> afd`. L'algorithme construira partiellement l'automate des parties (uniquement les états accessibles). Il sera nécessaire renuméroter les états accessibles pour éviter une explosion en mémoire de l'automate déterministe (on ne veut pas écrire un tableau de listes associatives de taille 2^{30}).
8. Construire avec ce qui précède un afd reconnaissant les mots finissant par *ababa*