Introduction aux langages

OPTION INFORMATIQUE - TP nº 3.7 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ utiliser le lemme de Levi
- ☞ manipuler un alphabet, un langage et ses puissances,
- ☞ programmer en OCaml des outils pour manipuler les langages.

A Mots, alphabets et lemme de Levi

On considère une alphabet Σ contenant au moins deux éléments.

A1. Soit Σ un alphabet. Soient a et b deux **lettres** de Σ. Montrer que $\forall u \in \Sigma^*, ua = bu \implies a = b$ et $u \in \{a\}^*$. On utilisera la définition inductive des mots.

Solution : On procède par induction en utilisant la définition inductive des mots.

(Cas de base) Soit $u = \epsilon$. Alors $ua = \epsilon a = a$ et $bu = b\epsilon = b$. Si $ua = bu$ alors on a $a = b$. De plus, comme $\epsilon \in \{a\}^*$, la propriété est vérifiée.

(Pas d'induction) On suppose maintenant qu'on dispose d'un mot u vérifiant la propriété $\mathcal{P}$: $ua = bu \implies a = b$ et $u \in \{a\}^*$. On cherche à construire v un mot à partir de u et à montrer que ce mot v vérifie également $\mathcal{P}$. Soit c une lettre de Σ. On peut construire un mot par la droite d'après la définition inductive des mots : soit $v = uc$. Alors on a $va = uca$. De même, $bv = buc$. Supposons que $va = bv$, c'est-à-dire $uca = buc$. Comme les lettres sont des éléments atomiques et la décomposition des mots uniques sur Σ, on a donc nécessairement : $a = c$ et $uc = bu$, c'est-à-dire $ua = bu$. Or, d'après notre hypothèse, u vérifie la propriété $\mathcal{P}$ et donc ceci implique de $a = b$ et $u \in \{a\}^*$. On a alors $va = uca = uaa$ et on en déduit que $v \in \{a\}^*$.

(Conclution) Quelque soit le mot u de Σ^*, la propriété $\mathcal{P}$ est vérifiée.

A2. Soient r, s, u, v et w quatre mots de Σ^* tels que $w = ur$ et $w = vs$. Montrer que u est un préfixe de v ou que v est un préfixe de u.

Solution : D'après le lemme de Levi, comme $ur = vs$, il existe un unique mot $z \in \Sigma^*$ tel que :

- soit $v = zu$
- soit $u = vz$

Quoiqu'il en soit, v est préfixe de u ou le contraire.

A3. Soient u et v deux mots de Σ^* qui vérifient $uv = vu$. Démontrer par induction structurelle sur les mots que :

$$\exists w \in \Sigma^*, \exists n, m \in \mathbb{N}, u = w^n \text{ et } v = w^m$$

Solution : On procède par induction sur la définition d'un mot. La propriété à vérifier est :

$$\mathcal{P} : \forall u, v \in \Sigma^*, uv = vu \Longrightarrow \exists w \in \Sigma^*, \exists n, m \in \mathbb{N}, u = w^n \text{ et } v = w^m$$

(Cas de base) Ce sont les cas où l'on considère le mot vide.

1. Si $u = v = \epsilon$ alors $w = \epsilon$, n'importe quelles valeurs de n ou de m conviennent.

(Pas d'induction) On suppose qu'on dispose de deux mots u et v qui vérifient $\mathcal{P}$. On construit un mot t à partir de u et on cherche à montrer que t vérifie $\mathcal{P}$.

Supposons que $uv = vu$. Par hypothèse d'induction, il existe donc deux entiers n et m tels que $u = w^n$ et $v = w^m$. Soit une lettre c de l'alphabet Σ. On peut construire par induction un mot $t = cu$ par la gauche. Supposons que $tv = vt$. Alors on a $cuv = vcu$. ce qui s'écrit $cw^{n+m} = w^m cw^n$. On peut simplifier par la droite par w^n. On obtient alors $cw^m = w^m c$ ce qui n'est possible que si $w = c$ (cf. résultat de la première question) puisque la décomposition d'un mot en lettres est unique (la lettre est atomique). On a donc $t = cu = w^{n+1}$. Le couple t et v vérifient la propriété $\mathcal{P}$. À partir de n'importe quel couple de mots vérifiant $\mathcal{P}$, la construction d'un autre mot par concaténation engendre un mot la vérifiant.

(Conclusion) La propriété $\mathcal{P}$ est vraie dans le cas de base et la règle de construction des mots ne modifie pas la propriété. $\mathcal{P}$ est donc vérifiée pour tout mots u et v de Σ^*.

A4. Soient deux mots u et v de Σ^*. Montrer que :

$$\exists p, q \in \mathbb{N}, u^p = v^q \Longleftrightarrow \exists w \in \Sigma^*, \exists n, m \in \mathbb{N}, u = w^n \text{ et } v = w^m$$

Solution :

($\Longleftarrow$) Soit p et q deux entiers. On a donc $u^p = w^{np}$ et $v^q = w^{mq}$. Il suffit donc de choisir $p = m$ et $q = n$ pour vérifier $u^p = v^q$.

($\Longrightarrow$) On a $u^p = uu^{p-1} = vv^{q-1} = v^q$. On applique le lemme de Lévi. Il existe alors un mot z de Σ^* tel que $vz = u^{p-1}$ et $zu = v^{q-1}$. On en déduit que $zuv = uvz$. D'après l'exercice précédent, il existe un mot w et deux entiers r et s tels que : $z = w^r$ et $uv = w^s$. Et donc il existe deux entiers n et m tels que $u = w^n$ et $v = w^m$.

A5. On définit les mots de Fibonacci sur l'alphabet $\Sigma = \{a, b\}$ par :

$$w_0 = \epsilon \quad (1)$$
$$w_1 = a \quad (2)$$
$$w_2 = b \quad (3)$$
$$w_n = w_{n-1} w_{n-2}, \forall n > 2 \quad (4)$$

(a) On suppose $n > 2$. Montrer que le suffixe de longueur deux de w_n est ba si n est impair et ab sinon.

Solution : Démonstration par récurrence sur n.

(b) On suppose $n > 3$ et on définit le mot v_n comme le préfixe de w_n obtenu en supprimant les deux dernières lettres. Montrer que v_n est un palindrome.

Solution : Par récurrence et en remarquant que $w_{n+1} = w_n w_{n-1} = v_n x y v_{n-1} y x$ avec $x = a$ et $y = b$ si n est pair, l'inverse sinon.
Le cœur de la démonstration s'appuie sur, si n est pair :

$$w_n = v_n ab = v_{n-1} ba v_{n-2} ab \tag{5}$$
$$w_{n+1} = w_n w_{n-1} = (v_{n-1} ba v_{n-2} ab v_{n-1}) ba \tag{6}$$

Comme par hypothèse de récurrence v_{n-1} et v_{n-2} sont des palindromes, alors $(v_{n-1} ba v_{n-2} ab v_{n-1})$ est un palindrome.

(c) Écrire une fonction récursive de signature `is_palindrome : string -> bool` qui permet de tester si une chaîne de caractères est un palindrome. Le pattern matching sur le type `string` n'est pas possible en OCaml car, contrairement aux listes, ce n'est pas type défini inductivement.

Solution :

```
let rec is_palindrome str =
    let n = String.length str in
        match n with
            | 0 | 1 -> true
            | _ -> if str.[0] = str.[n-1]
                   then is_palindrome (String.sub str 1 (n-2))
                   else false
    ;;

is_palindrome("non");;
is_palindrome("pas");;
is_palindrome("essayasse");;
is_palindrome("girafarig");;
is_palindrome("esope reste ici et se repose");;
is_palindrome("esoperesteicietserepose");;
```

(d) Écrire quatre version de la fonction signature `fib_word : int -> string` qui renvoie le nième mot de Fibonnacci. Ces quatre versions correspondent à :

1. une version à récursivité multiple,
2. une version à récursivité terminale,
3. une version itérative (programmation dynamique par le bas)
4. une version avec mémoïsation (programmation dynamique récursive).

Vérifier le résultat de la question b.

Solution :

```
let rec rec_fib_word n =
  match n with
    | 0 -> ""
    | 1 -> "a"
    | 2 -> "b"
    | _ -> rec_fib_word (n - 1) ^ rec_fib_word (n - 2);;

let ite_fib_word n =
  let u1 = ref "a" and u2 = ref "b" in
      match n with
        | 0 -> ""
        | 1 -> "a"
        | 2 -> "b"
        | _ ->  for _ = 3 to n do
                  let tmp = !u2 in
                  u2 := !u2 ^ !u1; u1:= tmp;
                done;
                !u2;;

let term_rec_fib_word n =
  let rec aux u2 u1 k = if k < n
                        then aux (u2 ^ u1) u2 (k + 1)
                        else u2 ^ u1
  in   match n with
      | 0 -> ""
      | 1 -> "a"
      | 2 -> "b"
      | _ -> aux "b" "a" 3
  ;;

  let memo_fib_word n =
      let memo = Hashtbl.create n in
      let rec aux k = match k with
                  | 0 -> ""
                  | 1 -> "a"
                  | 2 -> "b"
                  | _ -> if Hashtbl.mem memo k
                    then Hashtbl.find memo k
              else (Hashtbl.add memo k ((aux (k - 1)) ^ (aux (k - 2)))
                  ;
                    Hashtbl.find memo k;)
      in aux n;;



for i=0 to 9 do
  Printf.printf "rec_fib_word %i -> %s\n" i (rec_fib_word i);
  Printf.printf "term_rec_fib_word %i -> %s\n" i (term_rec_fib_word i)
      ;
  Printf.printf "ite_fib_word %i -> %s\n" i (ite_fib_word i);
  Printf.printf "memo_fib_word %i -> %s\n" i (memo_fib_word i);
  assert ((rec_fib_word i) = (ite_fib_word i));
  assert ((rec_fib_word i) = (term_rec_fib_word i));
  assert ((rec_fib_word i) = (memo_fib_word i));
```

```
    done;;

    for i=4 to 9 do
      let result = memo_fib_word i in
      Printf.printf "Fibonnacci word -> %s -> %b\n" result (is_palindrome(
          String.sub result 0 ((String.length result) - 2)));
      assert (is_palindrome(String.sub result 0 ((String.length result) -
          2)));
    done;;
```

B Langage et concaténation

B1. Soit $\mathcal{L}$ un langage sur Σ. Démontrer que $\mathcal{L}.\emptyset = \emptyset.\mathcal{L} = \emptyset$.

Solution : On utilise la définition de la concaténation d'un langage.

$$\begin{aligned} \mathcal{L}.\emptyset &= \{vw, v \in \mathcal{L} \wedge w \in \emptyset\} && (7)\\ &= \emptyset \text{ car } w \in \emptyset \text{ est impossible par définition} && (8) \end{aligned}$$

On procède de la même manière pour l'autre expression.

B2. Soit Σ un alphabet. Que vaut le cardinal de Σ^n en fonction du cardinal de Σ ? (S'appuyer sur la définition inductive de la puissance d'un langage)

Solution : On cherche à montrer que le cardinal de la puissance n d'un langage est le cardinal de ce langage à la puissance n. On fait donc l'hypothèse que $|\Sigma^n| = |\Sigma|^n$ On la démontre par induction en utilisant la définition inductive de la puissance d'un langage.

- Cas de base : pour $n = 0$, $|\Sigma^0| = |\{\epsilon\}| = 1 = |\Sigma|^0$
- Pas d'induction : pour $n \in \mathbb{N}^\star$, on suppose que $|\Sigma^n| = |\Sigma|^n$. Considérons maintenant :

$$\begin{aligned} |\Sigma^{n+1}| &= |\Sigma.\Sigma^n| && (9)\\ &= |\{vw, v \in \Sigma \wedge w \in \Sigma^n\}| && (10) \end{aligned}$$

Or, il n'y a que $|\Sigma|$ choix possible pour v. Il y a $|\Sigma|^n$ choix possibles pour w, par hypothèse d'induction. Donc, on a :

$$|\Sigma^{n+1}| = |\Sigma| \times |\Sigma|^n = |\Sigma|^{n+1} \qquad (11)$$

B3. On se donne l'alphabet `let sigma = ["a"; "b"; "c"]`, c'est à dire qu'on l'implémente par une liste. Écrire une fonction OCaml de signature `sigma_k : 'a list -> int -> 'a list list` qui génère le langage Σ^k sous la forme d'un liste de liste. Les éléments de cette liste seront les mots. Par exemple, `sigma_k sigma 2` renvoie :

```
[["a"; "a"]; ["a"; "b"]; ["a"; "c"]; ["b"; "a"]; ["b"; "b"]; ["b"; "c"]; ["c";
    "a"]; ["c"; "b"]; ["c"; "c"]]
```

Solution :

```
let sigma = ["a"; "b"; "c"];; (* Sigma *)

let rec sigma_k alphabet k =
    match k with
        | 0 -> []
        | 1 -> List.map (fun letter -> [letter]) alphabet
        | k -> List.fold_left
            (fun words letter ->
                words@(List.map (fun word -> letter::word) (sigma_k
                    alphabet (k - 1))))
            [] alphabet;;
sigma_k sigma 5;;
```

B4. Peut-on représenter Σ^* avec cette implémentation ?

Solution : Non, car le nombre d'éléments d'une liste est fini et Σ^* est un infini dénombrable. Ce que l'on démontre ci-dessous.

Démonstration. Soit Σ un alphabet à k lettres. On a montré précédemment que les ensembles Σ^n sont dénombrables puisque $|\Sigma^n| = |\Sigma|^n$. La fermeture de Kleene d'un ensemble est une union d'ensemble : $\Sigma^* = \bigcup_{n\geqslant 0} \Sigma^n$. Or, une union d'ensemble dénombrable est dénombrable.

On va le démontrer explicitement en exhibant une bijection entre Σ^* et $\mathbb{N}$.

On peut procéder de différentes manières :

- On peut décrire cette bijection ainsi : au mot vide, on associe 0. Puis on associe l'entier qui représente la position d'une lettre dans l'alphabet à cette lettre, $\forall i \in [\![1, k]\!], \forall a_i \in \Sigma, a_i \to i$. Pour chaque chaque mot de deux lettres triés dans l'ordre **lexicographique**, on associe les entiers suivants, il y en a k^2 : $\forall w \in \Sigma^2, w \to j \in [k+1, k+k^2]$. On procède de même pour la suite : $\forall w \in \Sigma^n, w \to j \in [\![\sum_{i=0}^{n-1} k^i, \sum_{i=1}^{n} k^i]\!]$
- On peut également procéder en attribuant directement un nombre au mot $a_{i_1} a_{i_2} \dots a_{i_n}$ le résultat de la fonction : $f(w) = \sum_{j=1}^{n} i_j k^{n-j}$. C'est ce que nous vons déjà fait dans le cadre des fonctions de hachage.

∎