Lexiques, Automates et Coroutines

ML appliqué au TAL

Gérard Huet

INRIA

Rocquencourt, Septembre 2001

Lexiques - Index

On code les lettres par des entiers. L'index d'un lexique est implémenté par un *trie*.

```
type code = list int;

type trie = [ Trie of (bool * list arc) ]
and arc = (int * trie);
```

Exemple. L'ensemble de codes $\{[1; 2], [2], [2; 2], [2; 3]\}$ est représenté comme:

```
Trie(False,[(1,Trie(False,[(2,Trie(True,[]))]));
            (2,Trie(True, [(2,Trie(True,[]));
                           (3,Trie(True,[]))]))])
```

Zippers - le retour

```
type zipper =
  [ Top
  | Zip of (bool * (list arc) * int * (list arc) * zipper)
  ]
and edit_state = (zipper * trie);
```

Un edit_state `(z,t0)` garde le contexte d'édition avec le zipper `z` et le subtrie courant `t0`. On remplace ce subtrie par le trie `t` en remontant le zipper par `zip_up t z`, avec
`zip_up: trie -> zipper -> trie` défini par :

```
value rec zip_up t = fun
  [ Top -> t
  | Zip(b,left,n,right,up) ->
    zip_up (Trie(b,rev_append left [(n,t)::right])) up
  ];
```

Deux routines de service

Soit $n : int$ et $l : listarcs$:. On construit `zip n l = (rev l1,l2)` avec l_1 max contenant des étiquettes $< n$:

```
value zip n = zip_rec []
  where rec zip_rec acc l = match l with
  [ [] -> (acc,[])
  | [((m,_) as current)::rest] ->
      if m<n then zip_rec [current::acc] rest
      else (acc,l)
  ];
```

Soit $c : code$. Le trie singleton le contenant est donné par `trie_of c` :

```
value rec trie_of = fun
  [ [] -> Trie(True,[])
  | [n::rest] -> Trie(False,[(n,trie_of rest)]) ];
```

Ajout d'un code à un trie

```
value enter code t = enter_edit code t Top
  where rec enter_edit c t z = match c with
      [ [] -> match t with
          [ Trie(_,l) -> zip_up (Trie(True,l)) z ]
      | [n::rest] -> match t with
          [ Trie(b,l) -> let (left,right) = zip n l
                         in match right with
            [ [] -> zip_up (trie_of rest) (Zip(b,left,n,[],z))
            | [(m,u)::r] ->
              if m=n then enter_edit rest u (Zip(b,left,n,r,z))
              else zip_up (trie_of rest) (Zip(b,left,n,right,z))
            ]
          ]
      ];
```

Appartenance d'un code à un trie

```
value rec mem_trie c = fun
  [ Trie(b,l) -> match c with
    [ [] -> b
    | [n::r] -> try let t = assoc n l in mem_trie r t
                with [ Not_found -> False ]
    ]
  ];
```

Lexiques - Associations

```
type trie_map 'a = [ Trie of (list 'a * list (arc_map 'a)) ]
and arc_map 'a = (int * trie_map 'a);

type zip_map 'a =
  [ Top
  | Zip of ((list 'a) * (list (arc_map 'a)) * int
                      * (list (arc_map 'a)) * (zip_map 'a))
  ];
```

Lexiques - Utilisation

Le traitement de la base de données lexicale produit :

- un index IR des mots-racine de type `trie`
- un lexique morphologique LM, de type `trie_map morph`

où `morph` définit la classe morphologique du mot-racine. Une machine grammaticale construit à partir de IM un lexique des formes fléchies LF, de type `trie_map flexed`, où `flexed` est une paire $(racine, flexion)$.

Quelques chiffres. Pour 12000 entrées, $|IR| = 100KB$. Pour 6000 racines lexicales, $|LM| = 250KB$, qui engendre 100000 formes fléchies dans $|LF| = 2.7MB$.

Memos - Programmation dynamique

```
type bucket = (int * list trie);

(* memo memorizes tries in buckets ordered by size *)
value memo = ref ([]:list bucket);

exception Absent of (* trie not yet stored in its bucket *)
   ( list bucket    (* inferior buckets *)
        * bucket    (*  proper  bucket  *)
   * list bucket)   (* superior buckets *);
(* returns a memo zipper for insertion by share *)

value register t (st,(n,l),r) =
  rev_append st [(n,[t::l])::r];
```

Memos - fetch et share

```
value fetch size trie = rem_rec [] memo.val
where rec rem_rec st m = match m with
   [ [((n,l) as p)::r] ->
      if n<size then rem_rec [p::st] r
      else if n=size then
             try search trie l
             with [Not_found -> raise (Absent(st,p,r))]
           else raise (Absent(st,(size,[]),m))
   | [] -> raise (Absent(st,(size,[]),m)) ];

value share trie size =
  try fetch size trie
  with [ Absent(z) -> (memo.val:=register trie z; trie)];
```

Lexiques - Partage en dag

```
(* make dag of trie by recognizing common subtries *)
value rec compress = fun
  [ Trie(b,arcs) ->
  let f (tries,size) (n,t) =
    let (t0,k) = compress t
    in ([(n,t0)::tries],size+k)
  in let (arcs0,s) = fold_left f ([],0) arcs
     in let t = share (Trie(b,arcs0)) s
        in (t,s+1)
  ];

value minimize trie = let (dag,_) = compress trie in dag;
```

Search

```
value search e = searchrec
  where rec searchrec = fun
   [ [] -> raise Not_found
   | [x::l] -> if x=e then x else searchrec l
   ];
```

Search is a basic list operation. Every good List library should have it.

Remark that *search* is equivalent to $search1$, with
`search1 e = find (fun x -> x=e)`,
in the sense that $\forall e, l\ search\ e\ l == search1\ e\ l$ but that it is NOT equivalent to $search2$, with
`search2 e l = if mem e l then e else raise Not_found`,
even though *search* and $search2$ are extensionally equal, i.e.:
$\forall e, l\ search\ e\ l = search2\ e\ l$.

Automates - les types en jeu

```
type lexicon = trie_map flexed
and rule = (code * code * code);
(*          (rev u, v    , w   ) such that u$v -> w   *)

type fsm = [ State of (bool * deter * choices) ]
          (* bool is True for accepting states *)
and deter = list (int * fsm)
and choices = list rule;

type stack = list choices; (* choice points stack *)
```

Automates - la construction

```
(* build_fsm : code -> lexicon -> (fsm * stack * int) *)
value rec build_fsm occ = fun
  [ Trie(d,arcs) ->
      let b = not (d=[]) (* True iff rev(occ) a flexed form *)
      in let local_stack = if b then get_sandhi occ else []
         in let f (deter,stack,size) (n,t) =
                let current = [n::occ]  (* current occurrence *)
                in let (auto,st,k) = build_fsm current t
                   in ([(n,auto)::deter],merge st stack,size+k)
            in let (deter,stack,size) =
                     fold_left f ([],[],0) arcs
               in let (h,l) = match stack with
                         [[] -> ([],[]) | [h::l] -> (h,l)]
                  in let s = share (State(b,deter,h)) size
                     in (s,merge local_stack l,size+1) ];
```

Automates - Compilation du lexique

```
exception Conflict;

value compile lexicon =
    let (automaton,stack,_) = build_fsm [] lexicon
    in if stack = [] then automaton else raise Conflict;
```

Automates - Le transducteur

```
type transition =
  [ Sandhi of (code * code * code) (* (rev u,v,w): u$v -> w *)
  | No_sandhi                      (* identity or no sandhi *)
  ]
and output = list (code * transition); (* code flexed form *)

type backtrack =
  [ Next of (input * output * code * deter * choices)
  | Init of (input * output)
  ]
and resumption = list backtrack; (* coroutine resumptions *)

exception Finished;
```

Automates - Routines de service

```
value rec prefix u v =
  match u with
    [ [] -> True
    | [a::r] -> match v with
        [ [] -> False
        | [b::s] -> a=b && prefix r s ]];
(* advance : int -> code -> code *)
value rec advance n l = if n = 0 then l
                        else advance (n-1) (tl l);
(* access : code -> fsm *)
value access = acc automaton
   where rec acc fsm = fun
      [ [] -> fsm
      | [c::rest] -> match fsm with
          [ State(_,deter,_) -> acc (assoc c deter) rest] ];
```

Le transducteur 1

```
value rec react input output back occ = fun
   [ State(b,det,choices) ->
     if b then
        let out = [(rev occ,No_sandhi)::output]
        in if input=[] then (out,back) (* opt final sandhi *)
           else let alterns =
                        [Next(input,output,occ,det,choices)::back]
                in react_init input out alterns
     else choose input output back occ det choices
   ]
and react_init input output back =
    react input output back [] automaton
```

Le transducteur 2

```
and choose input output back occ det = fun
  [ [] -> match input with
     [ [] -> backtrack back
     | [letter :: rest] ->
       try let next_state = assoc letter det
           in react rest output back [letter::occ] next_state
       with [ Not_found -> backtrack back ] ]
  | [((u,v,w) as rule)::others] ->
       let alterns = [Next(input,output,occ,det,others)::back]
       in if prefix w input then
             let tape = advance (length w) input
             and out = [((rev occ)@u,Sandhi(rule))::output]
             in let next_state = access v
                in react tape out alterns v next_state
          else backtrack alterns ]
```

Le transducteur 3

```
and backtrack = fun
    [ [] -> raise Finished
    | [resume::back] -> match resume with
        [ Next(input,output,occ,det,choices) ->
              choose input output back occ det choices
        | Init(input,output) -> react_init input output back
        ]
    ];
```

La coroutine de segmentation

```
value segment_first code = react_init code [] [];

(* resume : (resumption -> int -> resumption) *)
value resume cont n =
  let (output,resume) = backtrack cont
      in ( print_string "\n Solution "
         ; print_int n; print_string " :\n"
         ; iter print_out (rev output)
         ; resume
         );
```

Toutes les segmentations

```
value segment_all code = segment [ Init(code,[]) ] 1
  where rec segment cont n =
  try let resumption = resume cont n
      in segment resumption (n+1)
  with [ Finished ->
          if n=1 then print_string " No solution found\n"
          else () ];
```

Exemple

```
value process sentence = segment_all (code_string sentence);

process "avalokite\"svara";

 On input:
avalokite"svara

 We may segment:

 Solution 1 :
[  avalokita with sandhi a$ii -> e]
[  ii"svara with no sandhi]
```

Lemmatization

```
process "o.mnama.h\"sivaaya";
 Solution 1 :
[  om
 < und. form of om
 >  with sandhi m$n -> .mn]
[  namas
 < iic. form of namas#1
 | acc. sg. n. of namas#1
 | nom. sg. n. of namas#1
 | und. form of namas#1
 >  with sandhi s$"s -> .h"s]
[  "sivaaya
 < dat. sg. m. of "siva
 | dat. sg. n. of "siva
 >  with no sandhi]
```

Statistiques

On rappelle : LF comprend 100000 formes fléchies, avec $|LF| = 2.7MB$. Il y a 2790 règles euphoniques. Le calcul de l'automate depuis le lexique LF prend 7' sur petrus, y compris sa minimisation. On obtient un automate très compact, avec seulement 4845 états, dont 790 acceptants, occupant 438KB. 2822 états sont déterministes. Les 2023 autres ont un degré de branchement allant jusqu'à 164. Mais le nombre maximum de choix sur un état pour un input donné est de 2. Le segmenteur, en utilisation interactive, est instantané.

On peut espérer utiliser cette technologie dans des applications de reconnaissance de la parole.

Le papier, disponible sur `http://pauillac.inria.fr/~huet/PUBLIC/tagger.ps.gz`, donne les preuves de terminaison, correction et complétude de la méthode.