Automates mixtes

Gérard Huet

INRIA

12 mai 2003, LaBRI

Résumé

Les arbres lexicaux (tries) sont isomorphes aux espaces d'état des automates finis déterministes sans cycle. Leur partage en dag en produit la forme minimale. Des automates ou transducteurs d'états finis plus généraux peuvent être décrits par des arbres lexicaux annotés de décorations codant des transitions non déterministes vers des adresses virtuelles, des écritures de sortie, etc. En restant dans la problématique des types inductifs, ces structures de données pures peuvent être uniformément minimisées par le foncteur de partage, et donc donner lieu à des transducteurs minimaux. On obtient ainsi une représentation uniforme d'automates par des forêts de recouvrement de leurs graphes de transitions comme squelettes déterministes préférentiels minimaux. Ce point de vue est utilisé systématiquement dans la boîte à outils Zen, développée par l'auteur en Pidgin ML pour la programmation d'opérations phonologiques et morphologiques.

Automates simplistes

```
type letter = int
and word = list letter;

type trie = [ Trie of (bool × arcs) ]
and arcs = list (letter × trie);
```

Par exemple, voici l'arbre lexical représentant l'ensemble de mots {[1; 2], [2], [2; 2], [2; 3]} représentant {AB, B, BB, BC} :

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

Appartenance = Acceptance

```
value rec mem w = fun
  [ Trie(b,l) → match w with
    [ [] → b
    | [n::r] →
        try mem r (List.assoc n l)
        with [ Not_found → False ]
    ]
  ]; (* mem : word → trie → bool *)
```

Every trie defines the state graph of a *simplistic automaton*, with initial state the top node, and accepting states all nodes marked *True*. Acceptance of word w by automaton t is simply *mem w t*. Such automata are the acyclic deterministic finite-state automata, sharing initial common paths. They recognize the finite languages.

Minimal simplistic automata

Fact. Every simplistic automaton admits a unique equivalent minimal simplistic automaton.

Proof. Share the lexical tree as a dag, using the *Share* functor.

Algorithmic Remarks. Lists, binary trees, ternary trees. Zippers - unary contexts.

Important. Do not optimise too early.

Mixed automata

We add a zest of non-determinism, together with potential loops in the state space. But we want to stay in the applicative programming paradigm of inductive structures, and reject state graphs built with mutable references. We use a notion of virtual address, where states are named by a word defining a path from the initial state; such a word may not be unique, in case of sharing.

```
type address = [ Initial of word ];

type auto =  [ State of (bool × deter × choices) ]
and deter = list (letter × auto)
and choices = list (word × address);
```

Shortcomings

The nondeterministic part of a State represents a multiset of transitions guarded by a word. Since these guard words may be empty, we accommodate ϵ moves, but also general non-deterministic moves. The only problem is that virtual addresses must indeed point to locations accessible from the top node, whereas there exist non-deterministic automata which have no deterministic sub-automaton spanning the whole space set. In that case, we may still represent faithfully the non-deterministic automaton by considering a forest of trees, rather than a single tree.

Trie Forests

```
type address = [ Path of (int × word) ];
```

The rest of the construction is the same as above, but now an automaton is given as a vector of *auto* structures, together with the address of its initial state:

```
type auto =  [ State of (bool × deter × choices) ]
and deter = list (letter × auto)
and choices = list (word × address);

type automaton = (array auto × address);
```

Minimal mixed automata

Mixed automata have minimal representations as well, obtained uniformly by sharing their structure. However, in general such minimal representations may not be unique. Indeed, there may be other ways of representing an equivalent mixed automaton yielding smaller structures in terms of numbers of nodes and/or edges.

Furthermore, our simplistic virtual address mechanism will prevent the recognition of equivalent terminal sub-automata.

Bottom-up addressing

In order to facilitate such terminal sharing, and avoid the inefficient traversal of the state space from the root of the initial trees in the forest, we enrich our virtual addresses with local addresses, represented with *differential words*.

A differential word is a notation permitting to retrieve a word w from another word w' sharing a common prefix, as follows.

```
type delta = (int × word);
```

We compute the difference between w and w': as a differential word *(|w1|,w2)* where *w=p.w1* and *w'=p.w2*, with maximal prefix p. In ML, we compute *diff w w'*. And w' may be retrieved from w and *d=diff w w'* as *w'=patch d w*.

Mixed automata with duality

Differential words denote the shortest path between two nodes in the spanning tree of the structure, as an integer telling how many steps up is the closest common ancestor, and a word giving the path down. Now we authorize both global and local virtual addresses:

```
type address = [ Global of delta | Local of delta ];
```

The mixed automata type is defined as above, using this new *address* type. We now have duality between Global and Local addresses.

Using local addresses together with sharing raises the proper interpretation of such virtual addresses, since going up in a dag is not a well-defined notion. We need to keep a stack of accesses in the current deterministic space, while we traverse it. Taking it as a vector of *auto* nodes, global and local addresses are accessed similarly: indexing a vector of nodes, then navigating down a local trie.

A recognizer for mixed automata

```
type delta = (int × word)
and address = [ Global of delta | Local of delta ];

type auto =  [ State of (bool × deter × choices) ]
and deter = list (letter × auto)
and choices = list (word × address);

type automaton = (array auto × address);
```

An automaton value is a pair *(forest,init_address)* with *forest* an *auto* vector, and *init_address=Global(init_dag,init_path)*. We also declare an *auto* vector *local* of size greater than the maximum depth of the forest dags. The current state is represented as a natural number *depth* indexing the *local* vector.

Stack manipulation

Two operations on the stack index *depth* are provided: *pop*, which takes as argument a natural number, and *push*, which takes as argument a word:

```
value pop depth n =
  if n>depth then raise Stack_error else (depth-n);

value rec push depth = fun
  [ [] → depth
  | [letter :: rest] → let d=depth+1 in
      do { local.(d):=match local.(depth) with
             [ State(_,det,_) → assoc letter det ]
         ; push d rest }
  ];
```

Transitions

The next function executes a transition at a given address:

```
value transition depth = fun
  [ Global(n,w) → do {local.(0):=forest.(n); push 0 w}
  | Local(n,w) → push (pop depth n) w
  ];
```

We thus initialise the initial value of the stack from *init_address* by:

```
value init_depth = transition init_address;
```

and the initial state is *local.(init_depth)*, since we maintain as invariant that the current state is *local.(depth)*.

More service routines

The next service routine checks the prefix relation between words;
the following one advances the input tape by n characters;

```
value rec prefix u v =
  match u with
    [ [] → True
    | [a::r] → match v with
        [ [] → False
        | [b::s] → a=b && prefix r s
        ]
    ];
value rec advance n w = if n = 0 then w
                        else advance (n-1) (List.tl w);
```

The reactive engine

We represent the input as a word and a backtrack state as a triple storing a partial input, the depth indexing the current state in the stack, and a list of nondeterministic choices. Finally, a resumption is a set of backtrack states, which in a first approximation we represent as a list:

```
type input = word and depth = int;

type backtrack = (input × depth × choices)
and resumption = list backtrack;

exception Finished;
```

The reactive engine takes as arguments an input tape, a resumption, and the depth index in the *local* stack defining the current state.

React

```
value rec react input res depth = match local.(depth) with
  [ State(b,det,choices) →
    (* we try the deterministic space first *)
    let deter cont = match input with
        [ [] → backtrack cont
        | [letter :: rest] →
      try let next_state = List.assoc letter det in
          let next_depth = depth+1 in
          do { local.(next_depth):=next_state
             ; react rest cont next_depth }
      with [ Not_found → backtrack cont ]
        ] in
```

```
    let next_res = if choices=[] then res
                   else [(input,depth,choices) :: res] in
    if b then if input=[] then next_res (* solution *)
              else deter next_res
    else deter next_res
  ]
and backtrack = fun
  [ [] → raise Finished
  | [(input,depth,choices) :: res] →
    choose input res depth choices
  ]
and choose input res depth = fun
  [ [] → backtrack res
  | [(w,address) :: others] →
```

```
        let next_res = [(input,depth,others) :: res] in
         if prefix w input then
           let next_input = advance (List.length w) input
           and next_depth = transition depth address in
           react next_input next_res next_depth
        else backtrack next_res
  ];
```

Now, recognizing an input word is just calling the reactive engine on the appropriate initial situation:

```
value recognize w =
    try let _ = react w [] init_depth in True
    with [ Finished → False ];
```

Mixed transducers

It is easy to extend our mixed finite automata to mixed transducers, equipped with an output tape in addition to the input tape. Output words label the non-deterministic transitions.

```
type input = word and output = word and depth = int;

type trans =  [ State of (bool × deter × choices) ]
and deter = list (letter × trans)
and choices = list (input × output × address);

type transducer = (array trans × delta);

type backtrack = (input × output × depth × choices)
and resumption = list backtrack;
```

Transduce

The reactive engine above generalises in the natural way.

```
value transduce w =
  let (out,res) = react w [] [] init_depth in
  List.rev out;
```

This algorithm returns an output word (together with a resumption value) if its argument is recognized by the transducer; otherwise it raises the exception *Finished*.

Transducing coroutines

Methodological remarks about non-determinism programming.

Variations

The *local* vector holds the access stack in the current state component. We could also keep the sequence of letters defining the current transition context in a companion vector. This is useful for instance for generating output of copying transducers. It is shown for instance in the Zen manual how to recognize the language $L+$ of sequences of words from a lexicon L, with a transducer which outputs all "unglueing" solutions. Here the backtrack stack holds pairs *(input,output)*, where *output* keeps the local transition context. Another example is lemmatization. We may for instance store all plural forms in a trie, with accepting nodes holding as annotation the differential word addressing the singular form. Note that when we share this structure, all regular plurals are shared as one success node saying "my singular form is one level up on this access path".

From machines to patterns for reactive processes

More generally, we may define automata structures with various decorations, augmenting the acceptance boolean and the non-deterministic transitions with other values. And the reactive engine may recurse with other parameters. In this way we get away from the simple paradigm "automaton as a machine" to a more general paradigm "automaton state space as input pattern for a reactive process".

Conclusion

Modularity. Compiling regular expressions and relations.

Au revoir

Ma sabbatique à Bordeaux prend fin. Un grand merci au LaBRI de m'avoir accueilli.