Automata Mista

Gérard Huet

INRIA Rocquencourt,
BP 105, 78153 Le Chesnay Cedex, France,
Gerard.Huet@inria.fr,
http://pauillac.inria.fr/~huet

Abstract. We present a general methodology for non-deterministic programming based on pure functional programming. We construct families of automata constructions which are used as finite-state process descriptions. We use as algorithmic description language Pidgin ML, a core applicative subset of Objective Caml.

Dedicated to Zohar Manna for his 2^6th birthday

1 Introduction

We assume well-known the theory of finite-state machines, as presented in e.g. [11, 1, 5, 4]. Constructions of such machines, defined by compiling regular relations or other compositional formalism, typically alternate descriptions of deterministic automata and non-deterministic ones, using possibly ϵ moves, since the well known subset construction relate the two. Similarly, deterministic automata may be assumed minimal or not. Then more complex finite machine constructions may be obtained from high-level formalisms such as string rewrite rules or reactive programs.

However, in practice it is important to minimize the number of conversions between the various automata formats, as well as the minimization operations, if one wants to obtain tractable compilers on real-scale descriptions.

The traditional representation of the state space of finite automata and transducers is some kind of graph datatype, allowing arbitrarily crossing or looping transitions, but suffering from the dangers of explicit pointer reference mutation.

We shall give in this paper a number of purely applicative datatypes, and show how to use them for the description of finite-state machines in a way which facilitates reasoning, since induction over these structures is available, and which minimizes the cost of maintenance and debugging, since these datastructures are static. Furthermore such structures may be minimized as dags uniformly. We propose a formalism of *mixed transducers* (it. *automata mista*), which merge a deterministic skeleton with decorations by possibly non-deterministic transitions. Such transducers admit minimal representations, by shrinking their state space to a dag. We give applicative interpreters for such machines, which generate all possible transductions by resumption coroutines.

We give a termination argument for an important subclass of such finite machines, which uses the Dershowitz-Manna multiset ordering [3] in a natural way, and which is invariant by choice selection, allowing arbitrary selection tactics such as preference by weights obtained by training on annotated samples or other stochastic automata descriptions.

All our constructions use the algorithmic description language Pidgin ML, a core applicative subset of Objective Caml [10, 2]. A useful reference for the basic datatypes and algorithms is our description of the Zen toolkit [7, 9, 8].

2 Finite-state Machines

2.1 Simplistic Automata

We use as alphabet the natural numbers provided by the hardware processor:

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

Simplistic automata have their state graph expressed as a lexicon tree, or trie:

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

For instance, the trie storing the set of words {[1; 2], [2], [2; 2], [2; 3]} standing for the strings {AB, B, BB, BC} is represented as

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

Membership in a trie is expressed by a simple recursion:

```
(* mem : word → trie → bool *)
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 ]
    ]
  ];
```

Remark 1. We assume, but do not enforce, that for every letter there is at most one arc labeled with it from any trie node. We do not assume that arcs are listed in increasing order of the letters. More efficient implementations may use this assumption, implement the list of arcs as a binary search tree, etc. Many such variations exist, which may be justified by the actual application aimed at. As usual, we consider that such optimisations ought to be studied at a late stage in the design, after careful benchmarking on real-scale data. For the moment, think of our arcs a-lists as simple implementations of finite maps *letter* → *trie*.

Every trie t defines the state graph of a *simplistic automaton*, whose initial state is the top node, and accepting states are the trie nodes with a $True$ flag: $Trie(True, l)$. Acceptance of a word w by the simplistic automaton t is simply $mem\ w\ t$.

Simplistic automata are exactly the acyclic deterministic finite-state automata, with sharing of initial common paths. They recognize the finite languages.

2.2 Minimal Simplistic Automata

Proposition 1. *Every simplistic automaton admits a unique equivalent minimal simplistic automaton.*

Proof. Share the lexical tree as a dag, using the *Share* functor[6, 8].

Remark 2. The minimal simplistic automaton recognizing a given language represents the minimal deterministic finite-state automaton which recognizes its language. We leave the proof of this simple fact to the reader.

3 Mixed Automata

3.1 Non-deterministic Transitions

We now add a zest of non-determinism, together with potential loops in the state space. But since we want to stay in the applicative programming paradigm of inductive structures, we do not want to have 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 = [ Path of word ];

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

The nondeterministic part of an auto node represents a multiset of transitions guarded by a word. Since these guard words may be empty, we accommodate the traditional notion of ϵ move from finite automata folklore, but the general notion of a non-deterministic finite automaton as well. Indeed, the deterministic part is optional in this respect. 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. Let us assume that this forest is represented as an *auto* array. Since addresses may not be local to a tree, but may point to another tree in the forest, we shall represent them as a pair of an integer giving the tree index, and a word giving the local path in the indexed tree; thus we replace the address type above by:

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

The rest of the construction is the same as above, but now an automaton is given as an array 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);
```

3.2 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. Consider for example the automaton corresponding to the regular expression $AC^* + BC^*$. Representing it as (in the unique-tree setting)

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

does not allow any opportunity for sharing, even though the two internal nodes represent equivalent looping automata recognizing C^*, like in the recognizer for $(A + B)C^*$ represented as

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

where now sharing will factor the C^* automaton.

3.3 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* [6, 8].

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'*, where:

```
value rec diff = fun
    [ [] → fun x → (0,x)
    | [c :: r] as w → fun
        [ [] → (List.length w,[])
        | [c' :: r'] as w' →
            if c = c' then diff r r'
            else (List.length w,w')
        ]
    ];
```

Now *w'* may be retrieved from *w* and *d=diff w w'* as *w'=patch d w*, with:

```
value patch (n,w2) w =
  let p=truncate n (List.rev w) in unstack p w2;
```

where *truncate n w* is a list library function, truncating the initial prefix of length *n* from a word *w*, and *unstack* appends the reverse of its first argument to its second.

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, coupled with 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 automaton type is defined as above, using this new *address* type. Global plays the rôle of Path above, and its argument type is recognized as isomorphic to delta, but different operations apply to global and local addresses.

Thus the automaton above may now be represented as the tree:

```
State(False,[(1,State(True,[],[([3],Local(0,[]))]))
            ;(2,State(True,[],[([3],Local(0,[]))]))]);
```

which may then use a shared representation:

```
let loop = State(True,[],[([3],Local(0,[]))])
in State(False,[(1,loop); (2,loop)]);
```

But 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 state space, while we traverse it. This stack may be implemented as a list of *auto* nodes, or as an array of such values. In this last case we remark that global and local addresses are accessed by the same mechanism: indexing an array of state nodes, and then navigating down a local deterministic state space. For global addresses the array is the forest of mixed automata dags, for local addresses it is the local stack keeping the access context in the current dag. However, it is not clear that an array is better than a list, since when we store the state in a resumption set for non-determinism backtracking, we would have to copy the local access vector and not just store a reference to a unique mutable object. In the following for simplicity we choose the list implementation.

4 A Recognizer for Mixed Automata

4.1 Basic Operations

Let us sum up the current datatypes:

```
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);
```

The second component of an automaton is the (global) address of its initial state; by convention we could impose it to be $(0, [])$, i.e. the top node of the first dag in the array, but for ease of composing automata descriptions it is better to leave the generality of addressing any state node.

We assume that our automata are well-formed, in the sense that addresses are meaningful: global addresses index within the size of the automaton array, and local addresses index the stack within its depth.

We assume given an automaton value *(forest,init_address)* with *forest* an *auto* array, and *init_address=Global(init_root,init_path)*. The current state is represented as the current *auto* state node, plus the list *states* stacking the access path to it from the current root node.

Here is the way we access the automaton in its deterministic component, given an input letter:

```
value access state letter =
  match state with
    [ State(_,deter,_) → List.assoc letter deter ];
```

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

```
value rec pop state states n =
  if n=0 then (state,states) else match states with
     [ [] → raise Path_error
     | [ s :: up ] → pop s up (n-1)
     ];

value rec push state states = fun
  [ [] → (state,states)
  | [letter :: rest] →
    let new_state = access state letter in
    push new_state [state :: states] rest
  ];
```

The next function executes a transition defined by its address argument:

```
value transition state states = fun
 [ Global(n,w) → push forest.(n) [] w
 | Local(n,w) → let (s,ls) = pop state states n
                in push s ls w
 ];
```

Note that we assume that the automaton description is consistent, in the sense that all addresses are meaningful (i.e. access will never fail).

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);
```

4.2 The Reactive Engine

We represent the input as a word and a backtrack state as a tuple storing a partial input, the current state, the access stack of states, 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 states = list auto;

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

exception Finished;
```

The reactive engine takes as arguments an input tape, a resumption, the current state, and its access stack.

```
value rec react input res state states = match state with
  [ State(b,det,choices) →
    (* we try the deterministic space first *)
    let deter cont = match input with
          [ [] → backtrack cont
          | [letter :: rest] →
```

```
      try let state' = List.assoc letter det
          and states' = [ state :: states ] in
          react rest cont state' states'
      with [ Not_found → backtrack cont ]
         ] in
    let res' = if choices=[] then res
        else [(input,state,states,choices) :: res] in
    if b then if input=[] then res' (* solution *)
              else deter res'
    else deter res'
  ]
and backtrack = fun
  [ [] → raise Finished
  | [(input,state,states,choices) :: res] →
    choose input res state states choices
  ]
and choose input res state states = fun
  [ [] → backtrack res
  | [(w,addr) :: ch] →
    let res' = [(input,state,states,ch) :: res] in
    if prefix w input then
      let input' = advance (List.length w) input
      and (state',states') = transition state states addr
      in react input' res' state' states'
    else backtrack res'
  ];
```

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

```
value recognize w =
  let (init_state,init_states) =
      push forest.(init_root) [] init_path in
  try let _ = react w [] init_state init_states in True
  with [ Finished → False ];
```

It is an easy exercise to prove that *recognize* always terminate, provided there is no ϵ move in the automaton description, i.e. if every choice (w, a) is such that w is not empty. Furthermore, it will return *True* if and only if its argument belongs to the language generated by the automaton, using the standard notions. We shall actually see stronger versions of these properties below, when we extend the recognizer to a generator of transductions.

We remark that the automaton favors the deterministic transitions over the non-deterministic ones; this is significant for the following extension. We also remark that returning the resumption as the final result is useless when one is just interested in recognition, but is needed when one wants to enumerate all solutions, as we shall see.

5 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. We give the corresponding datatypes and algorithms below. The type *address* is the same as above, as well as the service routines *prefix*, *advance* and *transition*.

5.1 A Transducing Engine

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

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

type transducer = (array trans × address);

type backtrack = (input×output×trans×states×choices)
and resumption = list backtrack;

exception Finished;
```

The reactive engine takes as arguments an input tape, an output tape, a resumption, the current state and its access stack;

```
value rec react input output res state states =
  match state with
  [ State(b,det,choices) →
    let deter cont = match input with
          [ [] → backtrack cont
          | [letter :: rest] →
      try let state' = List.assoc letter det
          and states' = [ state :: states ] in
          react rest output cont state' states'
      with [ Not_found → backtrack cont ]
          ] in
    let res' =
      if choices=[] then res
      else [(input,output,state,states,choices) :: res] in
    if b then if input=[] then (output,res')
              else deter res'
    else deter res'
  ]
and backtrack = fun
```

```
  [ [] → raise Finished
  | [(input,output,state,states,choices) :: res] →
     choose input output res state states choices
  ]
and choose input output res state states = fun
  [ [] → backtrack res
  | [(inp,out,addr) :: ch] →
    let res' = [(input,output,state,states,ch) :: res] in
    if prefix inp input then
      let input' = advance (List.length inp) input
      and (state',states') = transition state states addr
      and output' = unstack out output in
      react input' output' res' state' states'
    else backtrack res'
  ];
```

Now, transducing an input word is just calling the reactive engine from the appropriate initial situation:

```
value transduce w =
  let (init_state,init_states) =
      push forest.(init_root) [] init_path in
  let (out,res) = react w [] [] init_state init_states
  in List.rev out;
```

This algorithm returns an output word if its argument is recognized by the transducer; otherwise it raises the exception *Finished*.

5.2 A Transducing Coroutine

Of course there may be several solutions to the transducing problem, and this is the rationale of the resumption component *res'* returned by *react*.

The process *backtrack* may thus be used in coroutine with a solution printer. Assuming a service routine *print_out* which prints solutions with their rank, we thus define

```
(* resume : resumption → int → resumption *)
value resume res n =
  let (output,res') = backtrack res
  in do { print_out n (List.rev output); res' };

value init_trans w =
  let (init_state,init_states) =
      push forest.(init_root) [] init_path in
  let (output,res) = react w [] [] init_state init_states
  in do { print_out 1 (List.rev output); res };

value transduce_all sentence =
```

```
try let res = init_trans sentence in restore res 2
    where rec restore res n =
    let res' = resume res n in
    restore res' (n+1)
with [ Finished → () ];
```

Many variations are possible. For instance, the *choose* routine may choose at wish among its choices multiset, instead of using a stack discipline. Indeed this choice may be guided by a priority weight associated with the non-deterministic transitions of the automaton, possibly computed by a training pre-processing; similarly, *backtrack* may choose among the backtrack multiset according to some selection strategy.

Also remark that we chose to make output transitions only on the non-deterministic arcs. This is in keeping with the view that the main part of our transducers consists in the deterministic automaton skeleton, but certain applications may warrant to allow output actions on the deterministic transitions as well. Also note that the output word is a stack, which should be reversed to yield the word result as a list of characters. Actually, the update

```
output' = unstack out output
```

could be replaced by a more informative construction of an output list documenting the output events and not just linearizing their trace:

```
output' = [out :: output]
```

but this introduces an asymmetry between input and output which suggests extending our construction to tree automata. We shall not develop further this remark in the current paper.

5.3 Termination

Definition 1. *If res is a resumption, we define* $\chi(res)$ *as the multiset of all* $\chi(back)$*, for back a backtrack value in res, where* $\chi(in, out, st, sts, ch) = \langle |in|, |ch| \rangle$.

χ *defines a well-founded ordering, with the standard ordering on natural numbers, extended lexicographically to triples for backtrack values and by multiset extension [3] for resumptions.*

We now associate a complexity to every function invocation. First $\chi(react\ in\ out\ res\ st\ sts) = \{\langle |in|, \kappa \rangle\} \oplus \chi(res)$*, where* $\oplus$ *is multiset union, and* $\kappa = 1 + |P|$*, with* P *the maximum number of non-deterministic transitions over all nodes. Then* $\chi(choose\ in\ out\ res\ st\ sts\ ch) = \{\langle |in|, |ch| \rangle\} \oplus \chi(res)$*. Finally we take* $\chi(backtrack\ res) = \chi(res)$.

We say that the transducer is strict iff there is no ϵ *move in its description, i.e. every choice* (w, a) *is such that* w *is not empty.*

Proposition 2. *If the transducer is strict, every call to backtrack*$(cont)$ *either raises the exception Finished, or else returns a value* (out, res) *such that* $\chi(res) < \chi(cont)$.

Proof. By nœtherian induction over the well-founded ordering computed by χ. It is easy to show that every function invocation decreases the complexity, we leave the details to the reader.

Corollary 1. *Under the strictness condition, resume always terminates, either raising the exception Finished, or returning a resumption of lower complexity than its argument. Therefore transduce_all always terminates with a finite set of solutions.*

Corollary 2. *Since we used a multiset complexity, invariant by permutation of the backtrack values in resumptions, we have actually proved the above results for a more abstract algorithm, where resumptions are not necessarily organized as sequential lists, but may be implemented as priority queues where elements are selected by an unspecified strategy or oracle. Thus these results remain for more sophisticated management policies of non-deterministic choices, obtained for instance by training on some reference annotated corpus.*

We believe the use of multiset ordering is an important general technique for proving the termination of non-deterministic processes.

5.4 Remarks

Sharing the state space consists in iterating bottom-up traversal with the *Share* functor [6] for all the roots of the forest. Remark that because of local addresses, such sharing goes beyond building an explicit smaller state graph, since local addresses are really description functions for the corresponding transitions, rather than mere pointers. This is reminiscent of the BDD techniques for representing succinctly boolean diagrams.

The variable *states* 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 path stack. This is useful for instance for generating output of copying transducers. It is shown for instance in [6] 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 (in a language such as French or English, where regular plurals just affix an 's' to their singular stem) are shared as one success node saying "my singular form is one level up on this access path".

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 a pattern for a reactive process".

An important extension of the present construction would be to abstract from the simple view of an automaton as a pair of its state space with the address of its initial state to a more modular view adding as third component a context continuation. There could be several continuation constructors: one used to start an automaton, another one to iterate the current automaton, the outer one to check that the input tape is exhausted. The reactive engine would get an extra argument, keeping the current automaton descriptor. On encountering the acceptance boolean, the current continuation would give rise to adding appropriate resumptions to the backtrack stack. This generalisation is crucial to the design of a modular compiler from regular expressions to mixed automata. We shall not develop this idea further in the current paper.

6 Applications

Finite-state methods are ubiquitous in computing; they are used in compilation, in circuit design, in computational linguistics, among other applications. In this last area, transducers are used to implement various regular relations in phonology and morphology toolkits. Our simplistic automata may be used to represent computer lexicons. The idea of differential words is used in [6] to represent in a compact way a flexed forms lexicon, in such a way that morphological functions are reversible, yielding directly a lemmatizer.

It is explained in [6] how to describe finite state automata and transducers as lexicon morphisms. As an instance, a sandhi analyser for the Sanskrit language is presented. It is shown that the method is sound and complete, and generates a finite number of solutions, using multiset ordering for its termination [3]. This example exhibits an impressive compression rate, since an automaton of 200000 states is shrunk to a shared structure of 7500 nodes.

We have abstracted from that work the present methodology of mixed automata and transducers. We believe that these structures are useful compromises between deterministic automata and non-deterministic transducers, amenable to uniform minimization by sharing, since their state spaces are completely applicative inductive datatypes.

Conclusion

We presented a methodology for describing finite-state automata and transducers in a completely applicative fashion, where a deterministic trie skeleton is decorated by attributes which store non-deterministic choice points, loops, and output transitions. There are no explicit references, and virtual addresses are of two kinds, bottom-up local references through an access stack, and top-down global accesses through the vector holding the state forest.

We believe that this structure is a useful compromise between completely deterministic, and general non-deterministic automata representations. In applications to computational linguistics, specially, there is a natural underlying main deterministic automaton, namely lexicon lookup. Many phonological and

morphological processes may benefit from the separation of concern between deterministic lexicon lookup and specific non-deterministic transducers implementing regular relations for phonological or morphological purposes.

We leave it to further research to define a language of regular expressions and relations, appropriate for the modular composition of mixed automata, and thus usable as a high-level interface for compiling finite-state processes into mixed transducers.

References

1. R. S. Alfred V. Aho and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
3. N. Dershowitz and Z. Manna. Proving termination with multiset ordering. *Commun. ACM*, 22:465–476, 1979.
4. M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, 1978.
5. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
6. G. Huet. Transducers as lexicon morphisms, phonemic segmentation by euphony analysis, application to a sanskrit tagger. `http://pauillac.inria.fr/~huet/FREE/tagger.pdf`, 2002.
7. G. Huet. The Zen computational linguistics toolkit. Technical report, ESSLLI Course Notes, 2002. `http://pauillac.inria.fr/~huet/ZEN/zen.pdf`
8. G. Huet. Linear contexts and the sharing functor: Techniques for symbolic computation. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*. Kluwer, 2003.
9. G. Huet. Zen and the art of symbolic computing: Light and fast applicative algorithms for computational linguistics. In *Practical Aspects of Declarative Languages (PADL) symposium, New Orleans*. LNCS 2562, Springer-Verlag, 2003.
10. X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. *The Objective Caml system, documentation and user's manual – release 3.00*. INRIA, 2000. `http://caml.inria.fr/ocaml/`
11. D. Perrin. Finite automata. In *Formal Models and Semantics. Handbook of Theoretical Computer Science, Volume B*. Elsevier and MIT Press, 1990.
12. E. Roche and Y. Schabes. *Finite-State Language Processing*. MIT Press, 1997.