Finite Eilenberg Machines

Benoît Razet

INRIA Paris-Rocquencourt,
benoit.razet@inria.fr

Abstract. Eilenberg machines define a general computational model. They are well suited to the simulation of problems specified using finite state formalisms such as formal languages and automata theory. This paper introduces a subclass of them called finite Eilenberg machines. We give a formal description of complete and efficient algorithms which permit the simulation of such machines. We show that our finiteness condition ensures a correct behavior of the simulation. Interpretations of this condition are studied for the cases of non-deterministic finite automata (NFA) and transducers, leading to applications to computational linguistics. The given implementation provides a generic simulation procedure for any problem encoded as a composition of finite Eilenberg machines.

Introduction

Samuel Eilenberg introduced in chapter 10 of his book [5], published in 1974, a notion of *Machine* which he claimed to be a *very efficient tool* for studying formal languages of the Chomsky hierarchy. They are sometimes referred to as *X-machines*. Many variants have appeared in the last twenty years [1] in several scientific domains different from formal languages.

Eilenberg machines define a general computational model. Assumed given an abstract data set X (it motivates X-machine terminology), a *machine* is defined as an automaton labelled with binary relations on X. Two generalizations result from this. Firstly, the set X abstracts the traditional tape used by automata on words, transducers *etc.* Secondly, compared to functions, binary relations give a built-in notion of non-determinism. Many translations of other machines into Eilenberg machines are given in [5]: automata, transducers, real-time transducers, two-way automata, push-down automata and Turing machines.

The remainder of this paper recalls the definitions of Eilenberg machines in Section 1. It also introduces a new subclass of them called *finite Eilenberg machines*. The Section 2 discusses the adaptation to the models of non-deterministic finite automata (NFA) and transducers which motivate their use for computational linguistics. The next two sections provide algorithms simulating finite Eilenberg machines. Since relations are central to Eilenberg machines, Section 3 proposes an encoding of them using streams. The Section 4 gives a formal description of algorithms which permit the simulation of finite Eilenberg machines in the spirit of the *reactive engine* introduced in [6].

1 Finite Eilenberg machines

We consider a monoid with carrier M, $\cdot$ an associative product on M and 1 its unit element. A finite monoid automaton $\mathcal{A}$ over M, also called a M-automaton, is a tuple (Q, δ, I, T) with Q a finite set of elements called *states*, δ a function from Q to finite subsets of $(M \times Q)$ called the *transition function*, I a subset of Q of *initial states* and T a subset of Q of *terminal states*.
A *path* p is a sequence $p = q_0 \xrightarrow{m_1} q_1 \xrightarrow{m_2} \cdots \xrightarrow{m_n} q_n$ with $n \in \mathbb{N}$ and $\forall i \leq n\ q_i \in Q$, $\forall i < n\ m_{i+1} \in M$ and $\forall i < n\ (m_{i+1},\ q_{i+1}) \in \delta(q_i)$. The path p is *successful* when $q_0 \in I$ and $q_n \in T$; its *length* is n. The *label* of p written $\bar{p}$ is 1 if $n = 0$ or $m_1 \cdot ... \cdot m_n$ otherwise. Finally, the *behavior* of the M-automaton $\mathcal{A}$, written $|\mathcal{A}|$, is defined as the set of all labels of successful paths of $\mathcal{A}$. We introduce the *type of* $\mathcal{A}$, written $\Phi_{\mathcal{A}}$, as the finite subset of M of elements appearing in the image of δ: $\Phi_{\mathcal{A}} = \{\ m \in M \mid \exists\ q\ q' \in Q,\ (m, q') \in \delta(q)\ \}$.

Let us now precise some notations for relations which are central to the remainder of this paper. A relation ρ from some set $\mathcal{D}$ to a set $\mathcal{D}'$ written $\rho : \mathcal{D} \to \mathcal{D}'$ is a set of pairs from $\wp(\mathcal{D} \times \mathcal{D}')$. The functional notation of its type is justified by the isomorphism between $\wp(\mathcal{D} \times \mathcal{D}')$ and $\mathcal{D} \to \wp(\mathcal{D}')$. The converse of a relation $\rho : \mathcal{D} \to \mathcal{D}'$ is written $\rho^{-1} : \mathcal{D}' \to \mathcal{D}$. Let us use $\rho(d)$ as notation for $\{\ d' \mid d' \in \mathcal{D}',\ (d, d') \in \rho\}$. The identity relation is written $id_{\mathcal{D}} = \{(d, d) | d \in \mathcal{D}\}$.

Let us recall Eilenberg's definition of machines. Let $\mathcal{D}$ be an arbitrary set called the *data* (it replaces the original notation X). We consider the set $R_{\mathcal{D}}$ of binary relations from $\mathcal{D}$ to $\mathcal{D}$. We consider here the *relations monoid* $R_{\mathcal{D}}$ with the *relation composition* $\circ$ as associative product and the *identity* relation id as unit element. A $\mathcal{D}$-*machine* $\mathcal{M}$ is a $R_{\mathcal{D}}$-automaton (Q, δ, I, T). With respect to the previous definitions the label of a path $p = q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \cdots \xrightarrow{\phi_n} q_n$ is the composition of relations $\bar{p} = \phi_1 \circ \cdots \circ \phi_n$. The behavior of $\mathcal{M}$ as an automaton, $|\mathcal{M}|$, is the set of relations of all labels of successful paths. The distinction between an automaton and a machine lies in the use of the union operation available on relations. The machine $\mathcal{M}$ defines a particular relation, written $||\mathcal{M}||$, as the relation union extended over all relations in $|\mathcal{M}|$:

$$||\mathcal{M}|| = \bigcup_{\rho \in |\mathcal{M}|} \rho\ .$$

We call the relation $||\mathcal{M}||$ the *characteristic relation* of the machine $\mathcal{M}$. We have given until now what we call the *kernel* of an Eilenberg machine which refers only to the automaton part.

The complete description of an Eilenberg machine requires what we call its *interface*. That is, consider $\mathcal{D}_-$ and $\mathcal{D}_+$ be two sets called respectively the *input* and *output* sets, an input relation $\phi_- : \mathcal{D}_- \to \mathcal{D}$ and an output relation $\phi_+ : \mathcal{D} \to \mathcal{D}_+$. Intuitively, the relation ϕ_- feeds the kernel with inputs and ϕ_+ interprets kernel results as outputs. A machine kernel with its interface defines a relation $\rho : \mathcal{D}_- \to \mathcal{D}_+$ as $\rho = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$. The usefulness of kernel and interfaces will be clear with examples provided in section 2.

Remark 1 (on modularity of Eilenberg machines). Any Eilenberg machine $\mathcal{M}$ of type $\Phi_{\mathcal{M}}$ defines a characteristic relation $||\mathcal{M}||$ that may belong to a type $\Phi_{\mathcal{M}'}$ of another Eilenberg machine $\mathcal{M}'$. This gives an idea that Eilenberg machines describe a modular computational model.

We are now going to introduce a new subclass of Eilenberg machines. For this purpose let us first define useful notions specific to machines rather than automata. Let us consider a $\mathcal{D}$-machine $\mathcal{M} = (Q, \delta, I, T)$. We call *cell* a pair $c = (d, q)$ of $\mathcal{D} \times Q$. An *edge* is a triple $((d, q), \phi, (d', q'))$, written $(d, q) \xrightarrow{\phi} (d', q')$ and satisfying the following two conditions $(\phi, q') \in \delta(q)$ and $d' \in \phi(d)$. A *trace* is a sequence of consecutive edges $t = c_0 \xrightarrow{\phi_1} c_1 \cdots \xrightarrow{\phi_n} c_n$. The integer n is the *length* of the trace. The cell c_0 is called its *beginning* and c_n its *end*. For each data d and state q, the cell (d, q) defines a null trace with itself as beginning and end. A cell (d, q) is said to be *terminal* whenever q is terminal. A trace t is said to be *terminal* when its end is terminal. Remark that each trace can be projected as the corresponding path when data are forgotten.

Definition 1.

1. *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two sets, we say that a relation $\rho : \mathcal{D}_1 \to \mathcal{D}_2$ is* ***locally finite*** *iff for all data d in $\mathcal{D}_1$ the set $\rho(d)$ is finite.*
2. *We say that a machine $\mathcal{M}$ is* ***locally finite*** *iff every relation ϕ in $\Phi_{\mathcal{M}}$ is locally finite.*
3. *The machine $\mathcal{M}$ is* ***globally finite*** *iff its characteristic relation $||\mathcal{M}||$ is locally finite.*
4. *The machine $\mathcal{M}$ is* ***nœtherian*** *iff there is no infinite trace $c_0 \xrightarrow{\phi_1} c_1 \ \cdots \ \xrightarrow{\phi_n} c_n \ \cdots$.*
5. *The machine $\mathcal{M}$ is called* ***finite*** *iff it is locally finite and nœtherian.*

Remark 2. A locally finite machine may or may not be globally finite and conversely a globally finite machine may or may not be locally finite.

Proposition 1. *If the machine $\mathcal{M}$ is finite then it is globally finite.*

Proof. Using König's lemma; the locally finite condition corresponds to the finite branching condition and the nœtherian condition to the non existence of infinite traces. □

Corollary 1. *Let ϕ_- and ϕ_+ be two partial functions. If the machine $\mathcal{M}$ is finite with interface ϕ_- and ϕ_+ then the relation $\phi_- \circ ||\mathcal{M}|| \circ \phi_+$ is locally finite.*

Let us now discuss the nœtherian condition. This definition with both the control and data may be arbitrarily complex. Of course an easy subcase is when there is no cycle in the automaton part of $\mathcal{M}$. Also it is easy to formulate a sufficient condition for a machine to be nœtherian:

Definition 2. *The machine $\mathcal{M}$ is of* ***nœtherian type*** *iff the relation $\bigcup_{\rho \in |\Phi_{\mathcal{M}}|} \rho$ is nœtherian.*

Proposition 2. *If the machine $\mathcal{M}$ is of nœtherian type then it satisfies the nœtherian condition.*

2 Examples and applications

We consider a finite set Σ of *letters* called the *alphabet*. We consider the free monoid Σ^* of *words* over Σ with the word concatenation as monoid product and the *empty word* ϵ as unit element. Formal languages are sets of words. Four basic operations on words are to be considered for defining Eilenberg machines for the next examples. For each letter σ of Σ:

- $L_\sigma = \{\ (w, \sigma w) \mid w \in \Sigma^*\ \}$, $L_\sigma^{-1} = \{\ (\sigma w, w) \mid w \in \Sigma^*\ \}$,
- $R_\sigma = \{\ (w, w\sigma) \mid w \in \Sigma^*\ \}$, $R_\sigma^{-1} = \{\ (w\sigma, w) \mid w \in \Sigma^*\ \}$.

The L and R denotations indicate operations respectively on the *left* or the *right* of a word. The last two relations are respectively the converse relations of the first ones. The *identity relation* on Σ^* is written id_{Σ^*}.

Remark 3. Relations L_σ, R_σ, L_σ^{-1}, R_σ^{-1} and id_{Σ^*} described above are in fact partial functions, thus they are locally finite relations.

Examples from Eilenberg show that his machine model implements many other computational paradigms. They use a notion of *"relabelling"* formally presented in [11] and recalled in Appendix **??**. We use them in the two following examples.

Example 1 (NFA). We consider here an alphabet Σ and words as elements of Σ^*. An NFA on alphabet Σ is a Σ^*-automaton $\mathcal{A}$ such that $\Phi_\mathcal{A} \subseteq \Sigma^*$ (ϵ-transitions are allowed). The set of words $|\mathcal{A}|$, the behavior of $\mathcal{A}$, is a formal language that belongs to the class of rational languages.

Let us define a relabelling procedure translating any NFA into an Eilenberg machine solving its word problem. Let $\mathcal{A} = (Q, \delta, I, T)$ be an NFA. We choose a data set $\mathcal{D} = \Sigma^*$. Since $\Phi_\mathcal{A} \subseteq \Sigma^*$, we recall the relabelling morphism α defined on Σ as $\alpha(\sigma) = L_\sigma^{-1}$ and then extended on Σ^*. Thus the machine $\mathcal{M}$ relabelled from $\mathcal{A}$ by α has the following characteristic relation: $||\mathcal{M}|| = \{\ (w\,w',\ w') \mid w \in |\mathcal{A}|,\ w' \in \Sigma^*\ \}$. That is, a given input $w{\cdot}w'$ is truncated by the word w recognized by the automaton $\mathcal{A}$. The encoding is completed with the following interface: $\mathcal{D}_- = \Sigma^*$, the input relation $\phi_- = id_{\Sigma^*}$, $\mathcal{D}_+ = \mathbb{B}$ the Boolean set composed of the two values $\top$ and $\bot$ and the output function $\phi_+ : \Sigma^* \to \mathbb{B}$ defined by $\phi_+(w)$ being $\top$ when $w = \epsilon$ and $\bot$ otherwise. Now we have $(\phi_- \circ ||\mathcal{M}|| \circ \phi_+)^{-1}(\top) = |\mathcal{A}|$. It shows that the relabelling is correct.

Let us now discuss the case when $\mathcal{M}$ is a *finite* Eilenberg machine. $\mathcal{M}$ is locally finite (Remark 3) and it satisfies the nœtherian condition whenever there is no ϵ-cycle in it. By ϵ-cycle we mean cycle labelled with id_{Σ^*}. Also $\mathcal{M}$ is of *nœtherian type* iff there is no ϵ-transition at all in it. It is true because for every edge $(u, q) \xrightarrow{L_w^{-1}} (v, q')$ we have $|w| > 1$ and then $|v| < |u|$, this shows that the length of traces is bounded by the length of their beginning word in their initial cell and thus there may not be infinite traces.

Example 2 (Transducers). Let Σ and Γ be two finite alphabets. The empty word ϵ will denote both empty words for Σ^* and for Γ^*. We consider here the monoid $\Sigma^* \times \Gamma^*$ with its traditional concatenation as associative product and

the pair (ϵ, ϵ) as unit element. A rational transducer from Σ^* to Γ^* is a monoid automaton $\mathcal{A}$ over $\Sigma^* \times \Gamma^*$ such that $\Phi_{\mathcal{A}} \subseteq (\Sigma \times \epsilon) \cup (\epsilon \times \Gamma)$. The subset of pairs of words from $|\mathcal{A}|$, the behavior of $\mathcal{A}$, defines a relation which belongs to the subclass of *rational relations*. Three problematics arise naturally :
– *Recognition* Given a couple of words (w, w') of $\Sigma^* \times \Gamma^*$, does (w, w') belong to $|\mathcal{A}|$.
– *Synthesis* Given a word w in Σ^* compute the set $|\mathcal{A}|(w)$ of words from Γ^*.
– *Analysis* Given a word w in Γ^* compute the set $|\mathcal{A}|^{-1}(w)$ of words from Σ^*.
For a given transducer these three problems may be encoded with the same automaton but using different relabellings and interfaces. The relabelling for the *synthesis* problem is defined as a morphism α on $\Phi_{\mathcal{A}}$ such that $\alpha(\sigma, \epsilon) = L_\sigma^{-1} \times id_{\Gamma^*}$ and $\alpha(\epsilon, \gamma) = id_{\Sigma^*} \times R_\gamma$. This encoding is completed with the following interface: $\mathcal{D}_- = \Sigma^*$, $\mathcal{D}_+ = \Gamma^*$, the input relation $\phi_- = \{ (w, (w, \epsilon)) \mid w \in \Sigma^* \}$ and the output relation $\phi_+ = \{ ((\epsilon, w'), w') \mid w' \in \Gamma^* \}$. Then we obtain $(\phi_- \circ ||\mathcal{M}|| \circ \phi_+) = |\mathcal{A}|$. $\mathcal{M}$ is locally finite (Remark 3) and satisfies the nœtherian condition whenever there is no cycle labelled with relation of $id_{\Sigma^*} \times R_\gamma$. $\mathcal{M}$ is of nœtherian type whenever there is no transition labelled with a relation of $id_{\Sigma^*} \times R_\gamma$ at all. As for NFA, this property is true with the same kind of argument concerning the length of the input tape.

Automata, transducers and more generally finite state machines are a popular technology for solving many computational linguistics problems [10]. We believe that the Eilenberg machines model is promising for this purpose. In fact, our restriction of *finite Eilenberg machines* is the formalism underlying the works concerning general morphological and phonetical modelings [6, 7] which have been applied to the Sanskrit language. Furthermore this application needs the modularity of such Eilenberg machine as sketched in the Remark 1. In the following we provide algorithms that simulate in a complete fashion any finite Eilenberg machine.

3 Streams representing relations

We consider now that the data set $\mathcal{D}$ is representable as an abstract ML datatype. We recall that ***unit*** is the singleton ML datatype containing the unique value denoted *()*. In our implementation we will use streams which are objects for enumerating on demand. In ML notation stream values are encoded with the following type parametrized with $\mathcal{D}$:

```
type stream D = | EOS
                | Stream of D × (delay D)
and delay D = unit → stream D ;
```

A stream value is either the empty stream *EOS* ("*End of Stream*") for encoding the empty enumeration or else a value *Stream d del* that provides the new element *d* of the enumeration and a value *del* as a delayed computation of the rest of the enumeration. Since ML computes with the restriction of λ-calculus to weak reduction, a value of type *delay* $\mathcal{D}$ such as *del* is delayed because it is

a functional value. This well known technique permits computation on demand. Note that this technique would not apply in a programming language evaluating inside a function body (strong reduction in λ-calculus terminology).

ML being a Turing-complete programming language, not all ML functions terminate and since our streams contain function values we shall restrict their computational power:

Definition 3.

1. *The ML function f:* $\textbf{unit} \to \alpha$ *is said to be total iff the evaluation of f () terminates (yielding a value of type* α*).*
2. *The ML stream str: stream* $\mathcal{D}$ *is said to be progressive iff*
   - *either str is EOS*
   - *or else str is of the form Stream d f with f total, and f () is progressive.*

We define the *head* function *hd* from non-empty streams to data defined as follows: $hd(Stream\ d\ del) = d$. We define also the *tail* function *tl* from streams to streams as $tl(EOS) = EOS$ and $tl(Stream\ d\ del) = del()$. Let n be an integer, we introduce the function tl^n that iterates the tl function n times:

$$\begin{cases} tl^0(str) & = str \\ tl^{n+1}(str) & = tl^n(tl(str)) \end{cases} \tag{1}$$

We introduce the predicate $InStream(d, str)$ that checks whether a data d appears in the stream str:

Definition 4. $InStream(d, str)$ *is true iff there exists an integer n such that* $(tl^n(str))$ *is non-empty and* $hd(tl^n(str)) = d$.

Definition 5. *A progressive stream str is finite iff there exists an integer n such that* $tl^n(str) = EOS$.

The *length* of a finite stream str, written $|str|$, is defined inductively as follows:

$$\begin{cases} |EOS| & = 0 \\ |Stream\ d\ del| & = 1 + |del()| \end{cases} \tag{2}$$

All finite streams of positive length end with a value of type *delay* $\mathcal{D}$ that associates to the unit element *()* the *EOS* stream announcing the end of the enumeration, typically:

```
value delay_eos = fun () → EOS;
```

We consider now relations of $R_{\mathcal{D}}$ representable as ML functions of the following type:

```
type relation 'D = 'D → stream 'D;
```

That is, if a relation *rel* of type *relation* $\mathcal{D}$ corresponds to a relation ρ of $R_{\mathcal{D}}$ then:

$$\forall d\ d' \in \mathcal{D},\ d' \in \rho(d) \Leftrightarrow InStream(d', rel\ d).$$

In the following we will use this technique only for representing locally finite relations. That is, relations are encoded using finite streams which are *progressive* by definition 5. From now on, we shall assume that our Eilenberg machines are *effective* in the sense that their data domain $\mathcal{D}$ are implemented as an ML datatype and that every relation used in their labeling is progressive.

4 A reactive engine for finite Eilenberg machines

We provide an implementation for the simulation of finite Eilenberg machines using higher-order recursive definitions. Algorithms are presented using ML notations which are directly executable in the OCaml programming language [8]. An essential feature of our formal notations is to possibly compose parametrized modules called functors. Algorithms are variants of the *reactive engine* [6]. They are presented completely using only a dozen of elegant definitions.

Let $\mathcal{M} = (Q, \delta, I, T)$ be a $\mathcal{D}$-machine. We specify $\mathcal{M}$ as a module with the following signature:

```
module type Kernel = sig
  type 𝒟 ;
  type state;
  value transition: state → list (relation 𝒟 × state);
  value initial: list state;
  value terminal: state → bool;
end;
```

The type parameter *state* encodes the set Q, the function *transition* encodes the function δ, the value *initial* encodes the initial states I as a list and the function *terminal* encodes the set of terminal states T as its characteristic predicate.

We aim at providing the algorithms implementing the characteristic relation of $\mathcal{M}$. For this purpose we use a functor that is a module parametrized by a *Kernel* machine. We call *Engine* this functor declared as the following:

```
module Engine (M: Kernel) = struct
  open M;
  ...(* body *)...
end;
```

Firstly, the body of the functor contains type declarations:

```
type choice = list (relation 𝒟 × state);
type backtrack =
  | Advance of 𝒟 × state
  | Choose of 𝒟 × state × choice × (delay 𝒟) × state
  ;
type resumption = list backtrack;
```

The type *choice* is an abbreviation for the list of transitions of the machine as used in the machine *M*. Eilenberg machines are possibly non-deterministic and need thus a backtracking mechanism for their implementation. Values of type *backtrack* allow to save the multiple choices due to the non-deterministic nature of the machine. The enumerating procedure will stack such backtrack values in a resumption of type *resumption*.

Secondly, the engine contains four functions. Three of them are internal and mutually recursive: *react*, *choose* and *continue*. They perform the non-deterministic search enriching the resumption as the computation goes on. The computation is performed in a depth-first search manner stacking the transition choices and streams within backtrack values in the resumption:

```
(* react: D → state → resumption → stream D *)
value rec react d q res =
  let ch = transition q in
  if terminal q
  then Stream d (fun () → choose d q ch res) (* Solution found *)
  else choose d q ch res

(* choose: D → state → choice → resumption → stream D *)
and choose d q ch res =
  match ch with
  | [] → continue res
  | (rel, q') :: rest →
    match (rel d) with
    | EOS → choose d q rest res
    | Stream d' del →
      react d' q' (Choose(d,q,rest,del,q') :: res)

(* continue: resumption → stream D *)
and continue res =
  match res with
  | [] → EOS
  | (Advance(d,q) :: rest) → react d q rest
  | (Choose(d,q,ch,del,q') :: rest) →
    match (del ()) with
    | EOS → choose d q ch rest
    | Stream d' del' →
      react d' q' (Choose(d,q,ch,del',q') :: rest)
;
```

The function *react* checks whether the state is terminal and then provides an element of the stream delaying the rest of the exploration calling to the function *choose*. This function *choose* performs the non-deterministic search over transitions, choosing them in the natural order induced by the ***list*** data structure. The function *continue* manages the backtracking mechanism and the enumeration of finite streams of relations, it always chooses to backtrack on the last pushed value in the resumption. Remark that these three mutually recursive functions do not use any side effect and are written in a pure functional style completely tail-recursive using the resumption as a continuation mechanism.

The machine $\mathcal{M}$ implemented as a module M has its characteristic relation $||\mathcal{M}||$ simulated by the following function:

```
(* characteristic_relation: relation D *)
value characteristic_relation d =
  let rec init_res l acc =
    match l with
    | [] → acc
    | (q :: rest) → init_res rest (Advance(d,q) :: acc)
  in continue (init_res initial []);
```

The function *characteristic_relation* first initializes the resumption with *Advance* backtrack values for each initial state and then call the function *continue* on it. We summarize the presented algorithms as follows: the machine $\mathcal{M}$ implemented as a module M has its characteristic relation $||\mathcal{M}||$ simulated by *Engine(M). characteristic_relation*, the function given by the instantiation of functor *Engine* with module *M*. We now provide the formalization with all arguments ensuring the correctness of our so-called reactive engine.

The formalization is inspired by the original one for the reactive engine [6]. We formalize the fact that data d and d' are in relation by the characteristic relation of $\mathcal{M}$ using the predicate $Solution(d, d')$ which is true iff there exists an initial state q and a terminal trace t beginning with cell (d, q) and ending with data d'. Now we give the correctness theorem of the reactive engine including its termination, its soundness and its completeness.

Theorem 1. *If the machine $\mathcal{M}$ is finite then characteristic_relation is a finite progressive relation (termination) and for all data d and d'*

$$InStream(d', characteristic\_relation\ d) \Leftrightarrow Solution(d, d').$$

Proof. The proof has been completely formalized and verified mechanically using the Coq proof assistant in [9]. Also a skeleton of the proof is provided in the appendix. It uses the *well-founded multiset ordering* technique as presented by Dershowitz and Manna [4] to prove termination. It also gives us a useful nœtherian induction principle used for the proofs of soundness and completeness (the two directions of the equivalence). □

Theorem 1 is a constructive version of Proposition 1. The reactive engine presented here *computes on demand* the solutions of a machine. This feature is important for combining in a modular fashion finite Eilenberg machines and keeping under control their evaluation.

5 Conclusion

Eilenberg machines provide a powerful and elegant framework for simulating specifications presented as finite automata variants. Eilenberg gave easy encodings into machines of formalisms at various levels of the Chomsky hierarchy. We have introduced a subclass of them called *finite Eilenberg machines* which are still general. They are *a priori* non-deterministic machines and we have shown that they behave as relations that associate to an input a **finite** number of computed outputs. Our machines are not restricted to treatments for the rational level of the Chomsky hierarchy. This particular point makes us believe that finite Eilenberg machines have applications to computational linguistics. In fact they are already efficient for explaining recognition or transduction problems that manipulate two levels of finite state formalisms such as explained in [7] for the modeling of the Sanskrit language. This multi-level ability is the modularity feature of Eilenberg machines. For this purpose implementations need to be

lazy. We anticipate future works in this spirit providing lazy algorithms. Our small but efficient *reactive engine* computes lazily the simulation of any finite Eilenberg machines. Our methodology using higher-order recursive definitions of functional programming language leads to formal proofs amenable to a complete formalization using higher-order logic. Such a formal development is available in the companion paper [9] using the Coq proof assistant [2]. Remark that this methodology leads to the same programs as shown here, by Coq's extraction mechanism from the formal development.

The enumeration of solutions implemented by the above algorithms uses a relatively naive lexicographic ordering. It is easy to refine these algorithms with more complex strategies, yielding weighted automata and stochastic methods such as *hidden Markov chains*. Such experiments will guide the design of the specification language for Eilenberg machines using regular expressions and compilation techniques such as presented in [3].

Acknowledgments

Gérard Huet and Jean-Baptiste Tristan took an essential part in the elaboration of this paper; I thank them for their participation.

References

1. Theory of X-machines. `http://www.x-machines.com`.
2. The Coq proof assistant. Software and documentation available on the Web, `http://coq.inria.fr/`, 1995–2007.
3. Cyril Allauzen and Mehryar Mohri. A unified construction of the Glushkov, Follow, and Antimirov automata. *In Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science (MFCS 2006)*, 4162 of Lecture Notes in Computer Science:110–121, 2006.
4. Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.
5. Samuel Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974.
6. Gérard Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional programming*, 15, 2005.
7. Gérard Huet and Benoît Razet. The reactive engine for modular transducers. In Jean-Pierre Jouannaud Kokichi Futatsugi and José Meseguer, editors, *Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 355–374. Springer-Verlag LNCS vol. 4060, 2006.
8. Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2006.
9. Benoit Razet. Simulating Eilenberg machines with a reactive engine: Formal specification, proof and program extraction. *INRIA Research Report*, 2008.
10. Emmanuel Roche and Yves Schabes. *Finite-state language processing*. MIT press, 1997.
11. Jacques Sakarovitch. Eléments de théorie des automates. *Vuibert, Paris*, 2003.