Simulation of Eilenberg Machines and Automata Synthesis

Gérard Huet & Benoît Razet

INRIA Rocquencourt

Séminaire à l'Université de Rouen
June 24^{th} 2009

Zen toolkit for computational linguistics

ZEN Toolkit

Zen is a computational linguistics toolkit developped for a Sanskrit processing platform:

- Written in the **OCaml** programming language.

ZEN Toolkit

Zen is a computational linguistics toolkit developped for a Sanskrit processing platform:

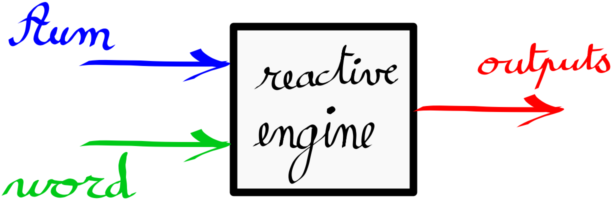
- Written in the **OCaml** programming language.
- It introduces the **Aum** data-structure for *"automata mista"* or *"mixed automata"*
  - Purely applicative data-structure.
  - States are adressed using a deterministic part.
  - Non-deterministic transitions and loops are encoded using virtual addresses.
  - Annotations for transductions, tagging...

ZEN Toolkit

Zen is a computational linguistics toolkit developped for a Sanskrit processing platform:

- Written in the **OCaml** programming language.
- It introduces the **Aum** data-structure for *"automata mista"* or *"mixed automata"*
  - Purely applicative data-structure.
  - States are adressed using a deterministic part.
  - Non-deterministic transitions and loops are encoded using virtual addresses.
  - Annotations for transductions, tagging...
- A reactive process called the **reactive engine** performs recognitions or synthesis or analysis...

An aum Interpreter

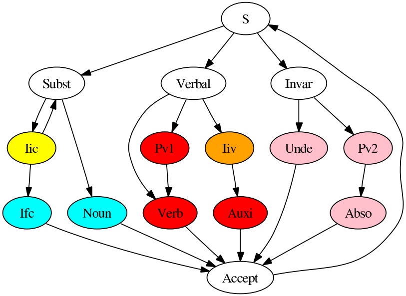


Sanskrit: two stages of Automata

- **Level 1**: Aums for lexicons: Noun, Pv, Verb, Unde, Auxi...

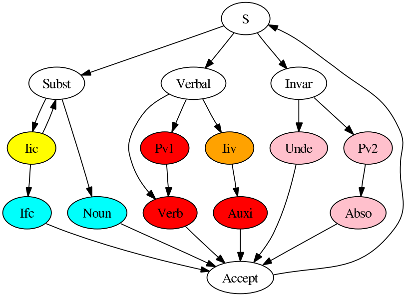
Sanskrit: two stages of Automata

- **Level 1**: Aums for lexicons: Noun, Pv, Verb, Unde, Auxi...
- **Level 2**: A NFA describing the morphology of Sanskrit words.



Sanskrit: two stages of Automata

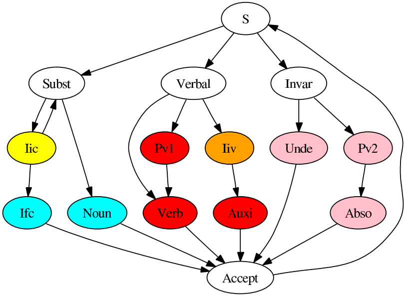
- **Level 1**: Aums for lexicons: Noun, Pv, Verb, Unde, Auxi...
- **Level 2**: A NFA describing the morphology of Sanskrit words.



- Now the reactive engine deals with 2 different automata controls.

Sanskrit: two stages of Automata

- **Level 1**: Aums for lexicons: Noun, Pv, Verb, Unde, Auxi...
- **Level 2**: A NFA describing the morphology of Sanskrit words.

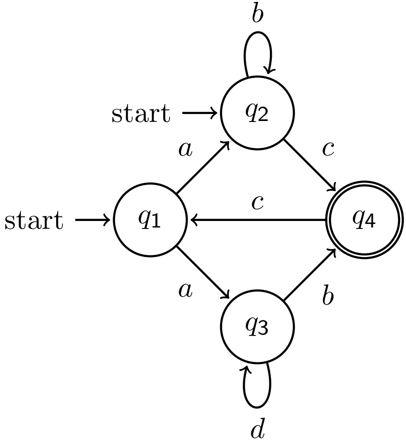


- Now the reactive engine deals with 2 different automata controls.

Idea! Each stage should be described as an instance of a unique model since they have the same nature.

The Eilenberg Machines Model

Built on Non-deterministic Finite Automata (NFA)



Monoid automata generalize NFA

Let $\mathcal{S} = (S, \cdot, 1)$ be a monoid, An $\mathcal{S}$-automaton $\mathcal{A} = (Q, \delta, I, T)$: Q finite set, δ function $Q \to \wp(S \times Q)$, $I \subseteq Q$, $T \subseteq Q$.
One defines:

- **path** : $p = q_0 \xrightarrow{s_1} q_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} q_n$
- **label of a path** : $lbl(p) = s_1 \cdot ... \cdot s_n$
- **valid path** : $vp(\mathcal{A})$, $q_0 \in I$ et $q_n \in T$
- The **behavior** of the automaton is the set of all labels of valid paths: $|\mathcal{A}| = \{lbl(p) \mid p \in vp(\mathcal{A})\}$.

Monoid automata generalize NFA

Let $\mathcal{S} = (S, \cdot, 1)$ be a monoid, An $\mathcal{S}$-automaton $\mathcal{A} = (Q, \delta, I, T)$: Q finite set, δ function $Q \to \wp(S \times Q)$, $I \subseteq Q$, $T \subseteq Q$.
One defines:

- **path** : $p = q_0 \xrightarrow{s_1} q_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} q_n$
- **label of a path** : $lbl(p) = s_1 \cdot \ldots \cdot s_n$
- **valid path** : $vp(\mathcal{A})$, $q_0 \in I$ et $q_n \in T$
- The **behavior** of the automaton is the set of all labels of valid paths: $|\mathcal{A}| = \{lbl(p) \mid p \in vp(\mathcal{A})\}$.

Two standard models of monoid automata:

$\mathcal{S} = \Sigma^*$	Σ^*-automaton	behavior = language
$\mathcal{S} = \Sigma^* \times \Gamma^*$	$\Sigma^* \times \Gamma^*$-automaton	behavior = relation on words

The Relational Model

Let $\mathcal{D}$ be an abstract set, for the data. A relation ρ from $\mathcal{D}$ to $\mathcal{D}$ is a subset of $\mathcal{D} \times \mathcal{D}$. A relation is considered as a model of non-deterministic computation.

The set of endo-relations, written $Rel(\mathcal{D})$, is a monoid:

- Composition : $\rho_1 \circ \rho_2 = \{ \ (x, z) \mid \exists y, \ x\rho_1 y \wedge y\rho_2 z \ \}$
- $Id = \{ \ (x, x) \mid x \in \mathcal{D} \ \}$
- $\langle Rel(\mathcal{D}), \ \circ, \ Id \rangle$ is a monoid.

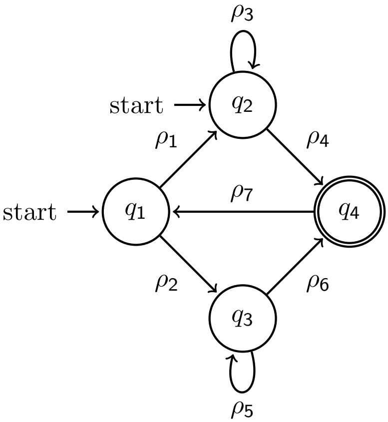
The Relational Model

Let $\mathcal{D}$ be an abstract set, for the data. A relation ρ from $\mathcal{D}$ to $\mathcal{D}$ is a subset of $\mathcal{D} \times \mathcal{D}$. A relation is considered as a model of non-deterministic computation.

The set of endo-relations, written $Rel(\mathcal{D})$, is a monoid:

- Composition : $\rho_1 \circ \rho_2 = \{\ (x, z) \mid \exists y,\ x\rho_1 y \wedge y\rho_2 z\ \}$
- $Id = \{\ (x, x) \mid x \in \mathcal{D}\ \}$
- $\langle Rel(\mathcal{D}),\ \circ,\ Id\rangle$ is a monoid.
- Union : $\rho_1 \cup \rho_2 = \{(x, y) \mid x\rho_1 y \vee x\rho_2 y\}$

Eilenberg Machines



Eilenberg Machines

$\mathcal{D}$ is an abstract set, for the *data*.
An Eilenberg Machine is a $Rel(\mathcal{D})$-automaton:

$$\mathcal{M} = (Q, \delta, I, T)$$

Eilenberg Machines

$\mathcal{D}$ is an abstract set, for the *data*.
An Eilenberg Machine is a $\mathit{Rel}(\mathcal{D})$-automaton:

$$\mathcal{M} = (Q, \delta, I, T)$$

From automaton structure we have :

- path : $p = q_0 \xrightarrow{\rho_1} q_1 \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_n} q_n$
- label of a path : $lbl(p) = \rho_1 \circ ... \circ \rho_n$
- valid path : $vp(\mathcal{M})$, $q_0 \in I$ et $q_n \in T$
- behavior : $|\mathcal{M}| = \{lbl(p) \mid p \in vp(\mathcal{M})\}$

Eilenberg Machines

$\mathcal{D}$ is an abstract set, for the *data*.
An Eilenberg Machine is a $Rel(\mathcal{D})$-automaton:

$$\mathcal{M} = (Q, \delta, I, T)$$

From automaton structure we have :

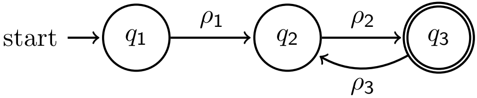
- path : $p = q_0 \xrightarrow{\rho_1} q_1 \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_n} q_n$
- label of a path : $lbl(p) = \rho_1 \circ ... \circ \rho_n$
- valid path : $vp(\mathcal{M})$, $q_0 \in I$ et $q_n \in T$
- behavior : $|\mathcal{M}| = \{lbl(p) \mid p \in vp(\mathcal{M})\}$

The **characteristic relation** of the machine $\mathcal{M}$ is the relation **union** of all labels of valid paths :

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

Example

Let $\mathcal{M}$ be the Eilenberg machine:

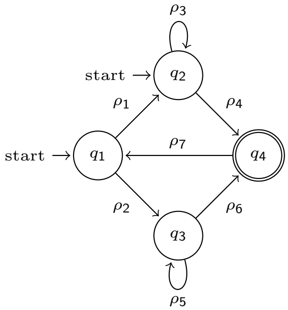


$$|\mathcal{M}| = \{\rho_1\rho_2,\ \rho_1\rho_2\rho_3\rho_2,\ \rho_1\rho_2\rho_3\rho_2\rho_3\rho_2,\ \cdots\}$$

$$||\mathcal{M}|| = \rho_1\rho_2\ \cup\ \rho_1\rho_2\rho_3\rho_2\ \cup\ \rho_1\rho_2\rho_3\rho_2\rho_3\rho_2\ \cup \cdots$$

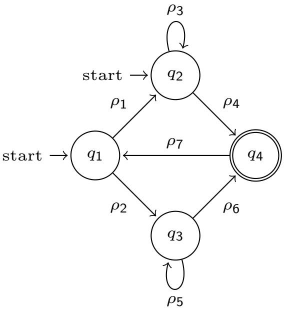
A **modular** computational model

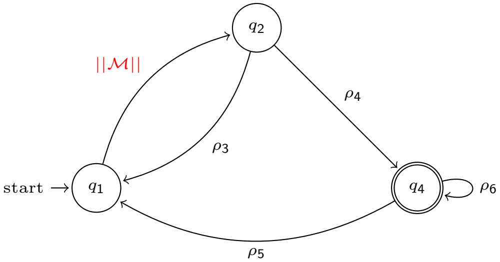
Let $\mathcal{M}$ be an Eilenberg machine, its characteristic relation $||\mathcal{M}||$ belongs to $Rel(\mathcal{D})$. Thus $||\mathcal{M}||$ can be used as a relation labelling another Eilenberg machine.



A **modular** computational model

Let $\mathcal{M}$ be an Eilenberg machine, its characteristic relation $||\mathcal{M}||$ belongs to $Rel(\mathcal{D})$. Thus $||\mathcal{M}||$ can be used as a relation labelling another Eilenberg machine.





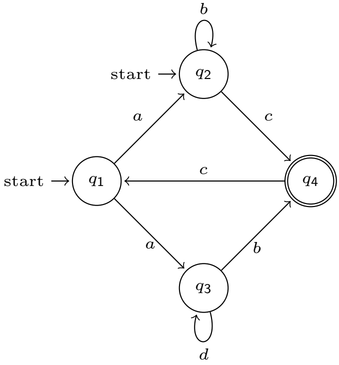
Automata, transducers, pushdown automata and Turing machines

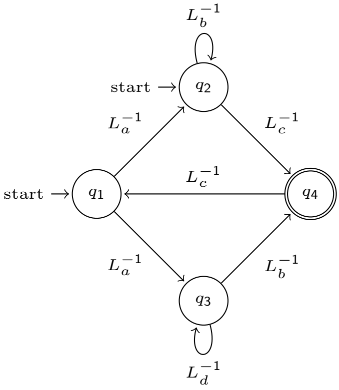
Automata, rational transducers, pushdown automata and Turing machines have in common a finite state control that uses tapes and stacks, on which they can read, write and move on... Let tapes be specified as data $\mathcal{D} = \Sigma^*$ then operations are partial functions from $\mathcal{D}$ to $\mathcal{D}$ and thus also as relations:

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

The NFA word acceptor as an Eilenberg machine

A word of a rational language L defined by an automaton is recognized by a machine $\mathcal{M}$ is simply obtained by a relabelling :





Then $||\mathcal{M}|| = \{(ww', w') \mid w \in L\}$. We refine $||\mathcal{M}||$ with a relation $\rho = \{(\epsilon, \epsilon)\}$:

$$||\mathcal{M}|| \circ \rho = \{(w, \epsilon) \mid w \in L\}$$

Finite automata, transducers, pushdown automata, Turing machines

What data domain $\mathcal{D}$?
What relations ρ labelling the machine?

	$\mathcal{D}$	ρ		
NFA	Σ^*	L_σ^{-1}		
ϵ-NFA	Σ^*	$L_{\sigma^\epsilon}^{-1}$		
Transducer	$\Sigma^* \times \Gamma^*$	$L_{\sigma^\epsilon}^{-1}$	$\times$	R_{γ^ϵ}
Realtime Trans	$\Sigma^* \times \Gamma^*$	L_σ^{-1}	$\times$	R_w
PDA (Pushdown)	$\Sigma^* \times \Gamma^*$	L_σ^{-1}	$\times$	$(L_\gamma^{-1} \cup L_\gamma)$
Turing Machines	$\mathbb{B}^* \times \mathbb{B}^*$	$(L_b^{-1} \cup L_b) \times Id_{\mathbb{B}^*}$	$\cup$	$Id_{\mathbb{B}^*} \times (R_b^{-1} \cup R_b)$

Samuel Eilenberg

Samuel Eilenberg, Marcel-Paul Schützenberger,
Seymour Ginsburg (ICALP 1972 at IRIA)

Simulation of Eilenberg Machines

Simulation ?

- Given a machine $\mathcal{M}$ and an "input" d of $\mathcal{D}$, we want to compute the set of solutions:

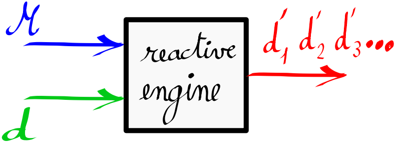
$$\{\ d' \mid d \xrightarrow{||\mathcal{M}||} d'\ \}$$

Simulation ?

- Given a machine $\mathcal{M}$ and an "input" d of $\mathcal{D}$, we want to compute the set of solutions:

$$\{ \ d' \mid d \xrightarrow{||\mathcal{M}||} d' \ \}$$

- Simulation adapting *Zen*'s **reactive engine**. The reactive engine enumerates the set of solutions.



Finite Eilenberg Machines

Let $\mathcal{M} = (Q, \delta, I, T)$, we define:

- **edge**: $(d, q) \xrightarrow{\rho} (d', q')$
  with $(\rho, q') \in \delta(q)$ and $d' \in \rho(d)$.
- **path**: $p = (d_0, q_0) \xrightarrow{\rho_1} (d_1, q_1) \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_n} (d_n, q_n)$

Definition (Finite Eilenberg Machines)

1. *Locally finite condition: For all relation ρ labelling $\mathcal{M}$, ρ is a locally finite relation: for all data d, the set $\rho(d)$ is finite.*
2. *Nœtherian condition: The length of any path is necessarily finite.*

$$(d_0, q_0) \xrightarrow{\rho_1} (d_1, q_1) \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_n} \cdots$$

Proposition (Koenig's Lemma)

*The characteristic relation $||\mathcal{M}||$ is a **locally finite relation**.*

About the Nœtherian condition

There are two cases for which the Nœtherian condition is satisfied :

- The state graph contains no cycle : the length of paths is bounded by the length of the automaton path.
- There is a Nœtherian relation $>$ on $\mathcal{D}$ such that for all relation ρ of the machine, for all data d and d',

$$d' \in \rho(d) \Rightarrow d > d' \ .$$

About finite automata as finite Eilenberg machines

First, relations are always locally finite.

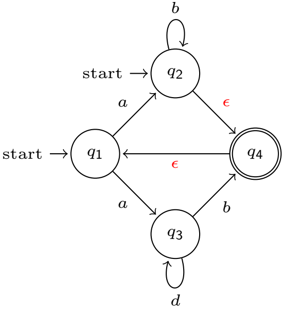
But the second condition shall be discussed:

- DFA OK. (The tape decreases after each transition)
- NFA OK. (The tape decreases after each transition)
- ϵ-NFA It depends.

About finite automata as finite Eilenberg machines

First, relations are always locally finite.
But the second condition shall be discussed:

- DFA OK. (The tape decreases after each transition)
- NFA OK. (The tape decreases after each transition)
- ϵ-NFA It depends.
  without ϵ-cycle OK.



Design choices

For simulating Eilenberg machines in a programming language we need:

- Polymorphism: for $\mathcal{D}$, the abstract data domain.
- Relations of $Rel(\mathcal{D})$ may be seen as functions thanks to the following isomorphism:

$$\rho \in \wp(\mathcal{D} \times \mathcal{D}) = \mathcal{D} \rightarrow \wp(\mathcal{D})$$

- Finite sets are enumerated using *streams*

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

type relation D = D -> (stream D);
```

- Higher-order constructions: Eilenberg machines are automata labelled with relations.

Correctness of the reactive engine

Theorem (Soundness and Completeness)

Let `M : Machine` *be a finite Eilenberg machine.* $\forall d\ d' \in \mathcal{D}$,
$d' \in ($`reactive_engine M`$\ d) \Leftrightarrow$ `Solution M d d'` .

Correctness of the reactive engine

Theorem (Soundness and Completeness)

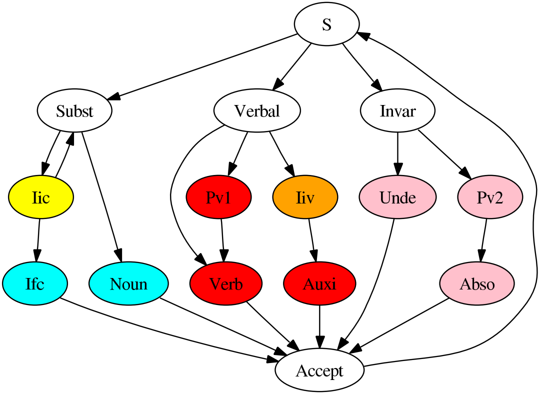
Let `M : Machine` *be a finite Eilenberg machine.* $\forall d\ d' \in \mathcal{D}$, $d' \in (\texttt{reactive\_engine}\ M\ d) \Leftrightarrow$ `Solution M d d'` .

Formally proved in **Coq**

Example 1: Modularity

A Sanskrit segmenter with 2 stages of automata:

- A NFA for the geometry of a Sanskrit word
- Aums for lexicons



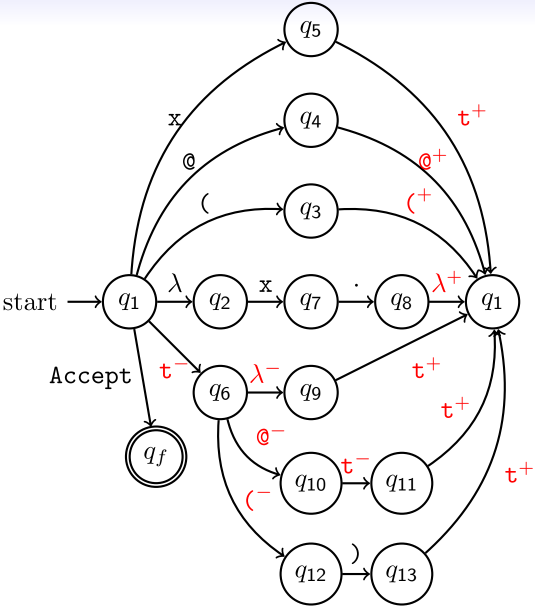
Example 2: a non-deterministic model simulated completely

A complete backtracking parser for an ambiguous grammar for λ-calculus.

Consider the following ambiguous grammar:

$$\begin{array}{cccc} T & := & \mathtt{x} & \text{(variable)} \\ & | & \lambda\mathtt{x}.T & \text{(abstraction)} \\ & | & T\,\mathtt{@}\,T & \text{(application)} \\ & | & \mathtt{(}T\mathtt{)} & \end{array}$$

Following this grammar the λ-term `"`λ`x.x@`λ`x.x"` may be recognized as `"`λ`x.(x@`λ`x.x)"` but also as `"(`λ`x.x)@(`λ`x.x)"`.


start
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
q_{11}
q_{12}
q_{13}
q_f
x
@
(
λ
.
Accept
$\mathtt{t}^+$
$\mathtt{@}^+$
$\mathtt{(}^+$
λ^+
$\mathtt{t}^-$
λ^-
$\mathtt{@}^-$
$\mathtt{(}^-$
)

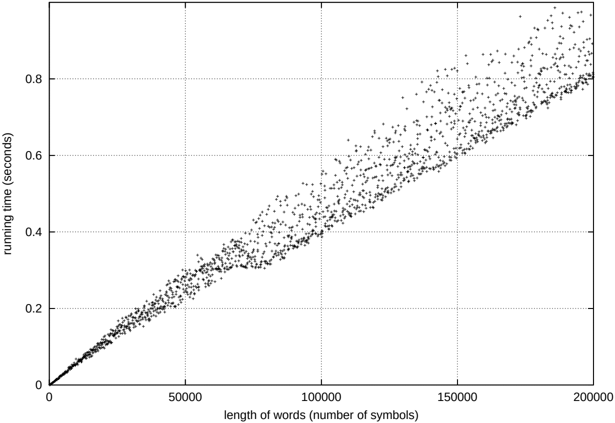
Benchmarks

Finding all Solutions:

- `"`λ`x.x@(`λ`x.`λ`x.x@x)@x@x@`λ`x.x@x"`: 522 solutions instantaneously.
- `"x@x@x@x@x@x@x@x@x@x@x@x"`: 208012 solutions : 9 seconds of running time.

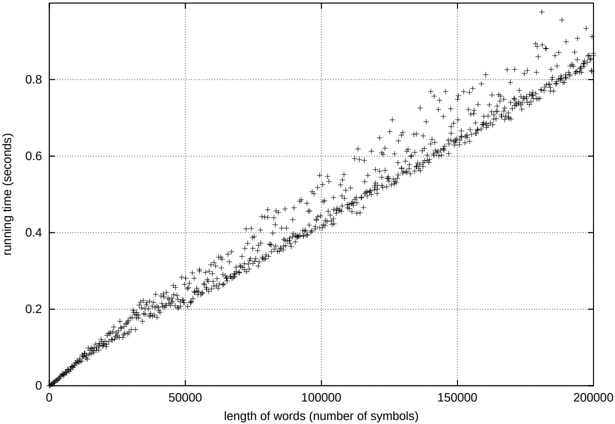
The first solution

For randomly generated *ambiguous* λ-terms:



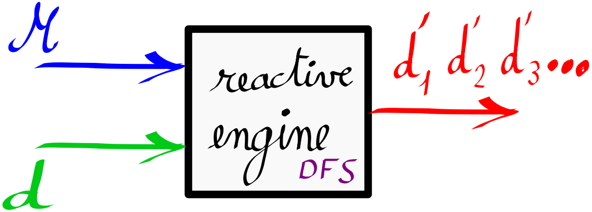
All solutions

For randomly generated *unambiguous* λ-terms (with all parentheses):

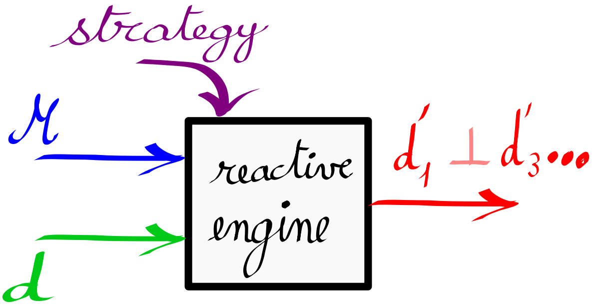


Towards computable Eilenberg machines

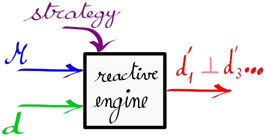
The reactive engine for finite Eilenberg machines uses a built-in *depth-first search* strategy :



Towards computable Eilenberg machines

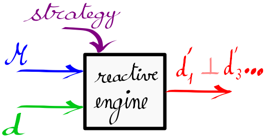


Towards computable Eilenberg machines



- The strategy could be
  - Depth-first search (finite Eilenberg Machines)
  - Breadth-first search
  - Deterministic strategy: generalization of deterministic automata DFA
  - Cantor enumeration: One particular complete strategy
  - Fair strategies

Towards computable Eilenberg machines



- The modularity needs more general streams: Recursively enumerable sets are enumerated using *streams*

```
type stream 𝒟 =
  | Done
  | Elm of 𝒟 × (delay 𝒟)
  | Skip of delay 𝒟
and delay 𝒟 = unit -> (stream 𝒟);

type relation 𝒟 = 𝒟 -> (stream 𝒟);
```

From regular expressions to automata

Regular expressions for regular languages

$$\begin{array}{rcc} E, F & := & \mathsf{0} \\ & | & \mathsf{1} \\ & | & a, a \in \Sigma \\ & | & E + F \\ & | & E \cdot F \\ & | & E^* \end{array}$$

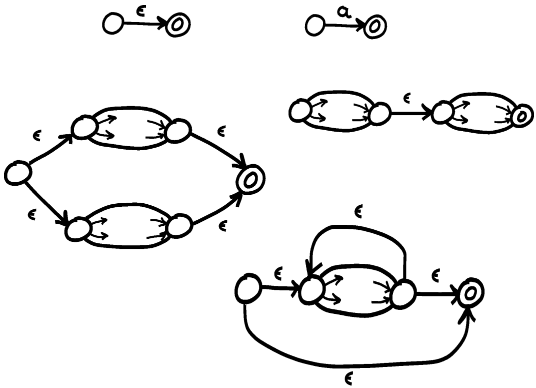
Theorem (Kleene 1956)

$$\forall \mathcal{A}, \exists E, L(\mathcal{A}) = L(E),$$
$$\forall E, \exists \mathcal{A}, L(E) = L(\mathcal{A})$$

Thompson's algorithm (1968)

Recursive algorithm over the expression, producing an ϵ-NFA in a unique traversal.

$$1,\ a,\ E+F,\ E\cdot F,\ E^*$$



Thompson's algorithm (1968)

```
value thompson e =
  let rec aux e t n = (* e is current regexp, t accumulates the state space, n is last created location *)
    match e with
    [ One ->
      let n1=n+1 and n2=n+2 in
      (n1, [ (n1, [ (None, n2) ]) :: t ], n2)
    | Symb s ->
      let n1=n+1 and n2=n+2 in
      (n1, [ (n1, [ (Some s, n2) ]) :: t ], n2)
    | Union e1 e2 ->
      let (i1,t1,f1) = aux e1 t n in
      let (i2,t2,f2) = aux e2 t1 f1 in
      let n1=f2+1 and n2=f2+2 in
      (n1, [ (n1, [ (None, i1); (None, i2) ]) ::
                [ (f1, [ (None, n2) ]) ::
                [ (f2, [ (None, n2) ]) :: t2 ] ] ], n2)
    | Conc e1 e2 ->
      let (i1,t1,f1) = aux e1 t n in
      let (i2,t2,f2) = aux e2 t1 f1 in
      (i1, [ (f1, [ (None, i2) ]) :: t2 ], f2)
    | Star e1 ->
      let (i1,t1,f1) = aux e1 t n in
      let n1=f1+1 and n2=f1+2 in
      let t1' = [ (f1, [ (None, i1); (None, n2) ]) :: t1 ] in
      (n1, [ (n1, [ (None, i1); (None, n2) ]) :: t1' ], n2)
    ] in
  aux e [] 0
;
```

From regular expressions to automata

Automaton	Algorithm	Complexity	Type
Thompson	Thompson(1968)	$O(n)$	ϵ-NFA
Position	Berry-Sethi (1986)	$O(p^2)$	NFA
Follow	Ilie & Yu (2003)	$O(p^2)$	NFA
Equation	Antimirov (1996)	$O(p^2)$	NFA

Size comparison (states):

Position > Follow > Equation

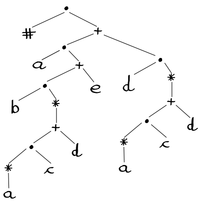
Position, Follow & Equation automata

The algorithms proceeds in 2 successive steps:

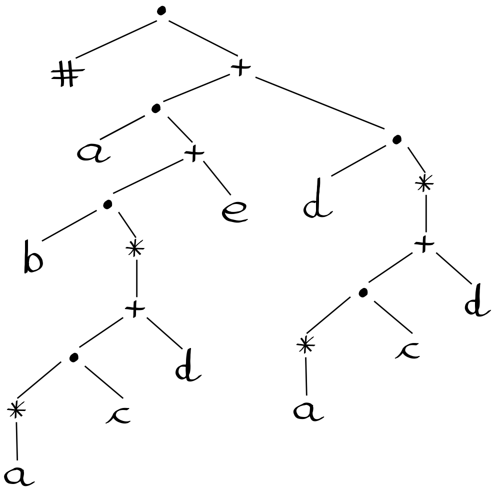
1. Identify the states
2. Compute the transitions

Example

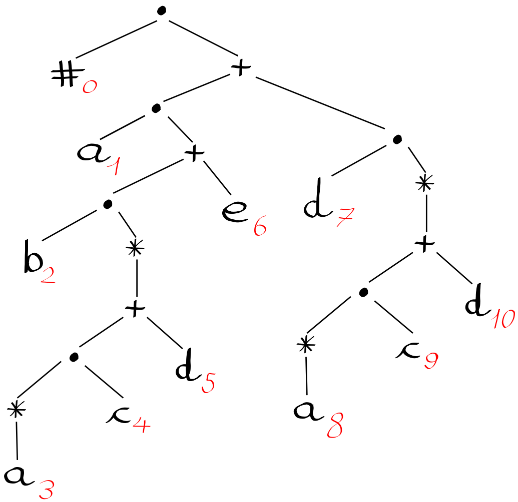
$E = a(b(a^*c + d)^* + e) \quad + \quad d(a^*c + d)^*$



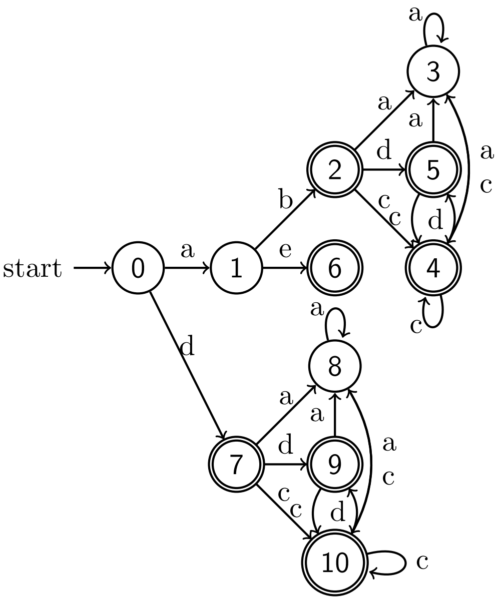
Position automaton



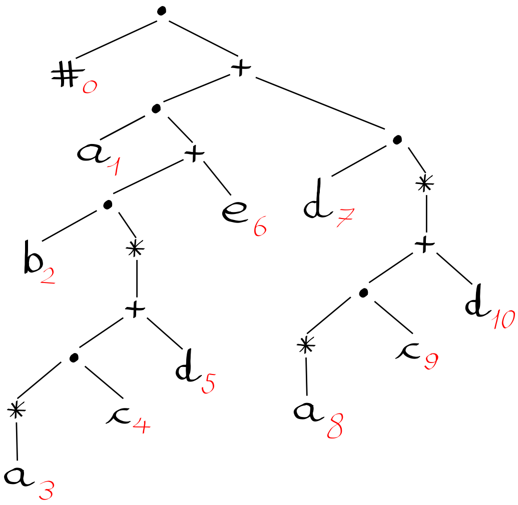
Position automaton



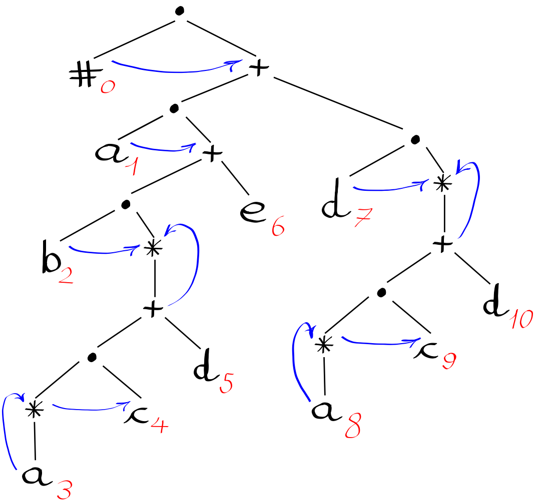
Position automaton



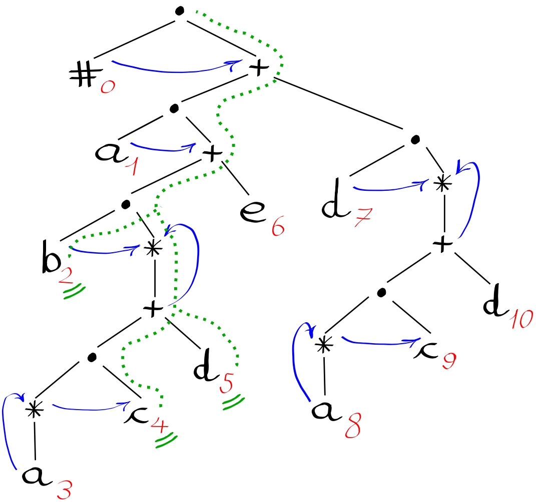
Follow automaton



Follow automaton

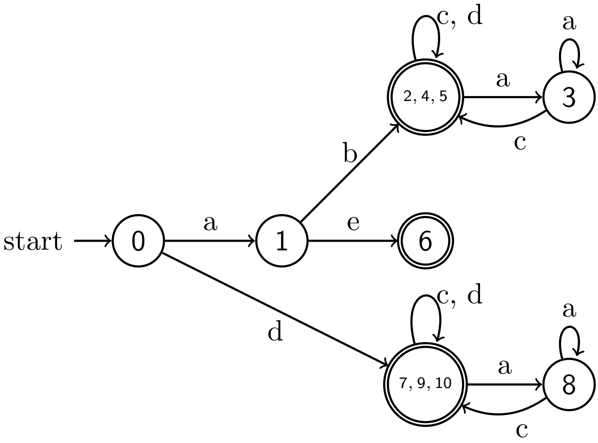


Follow automaton



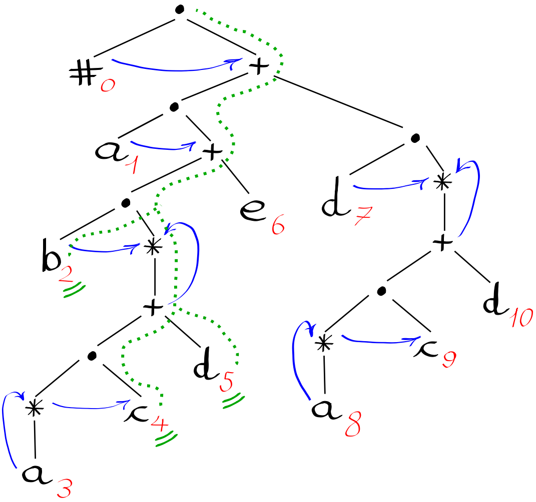
Follow automaton

$2 = 4 = 5$

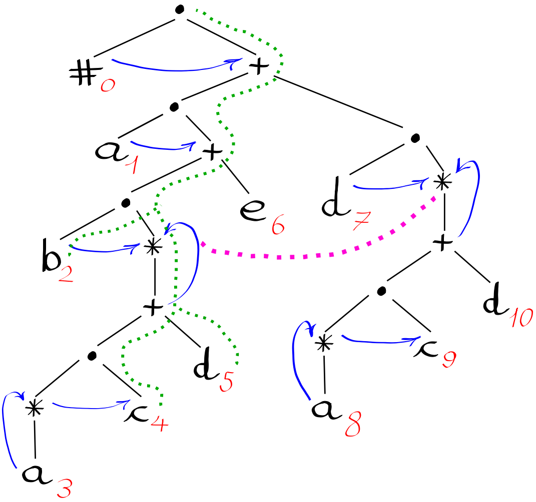


$7 = 9 = 10$

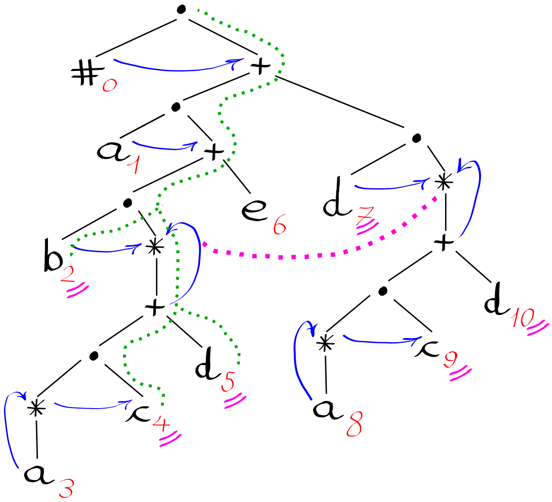
Equation automaton



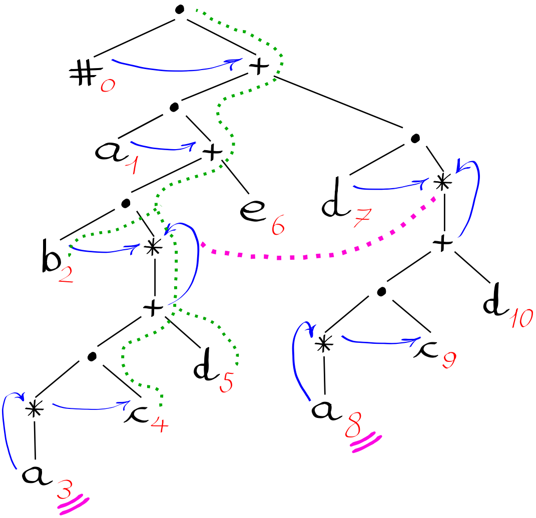
Equation automaton



Equation automaton

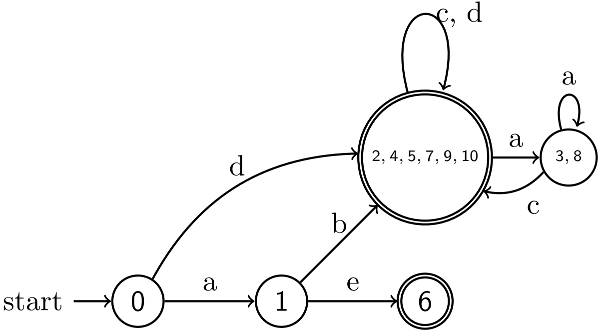


Equation automaton



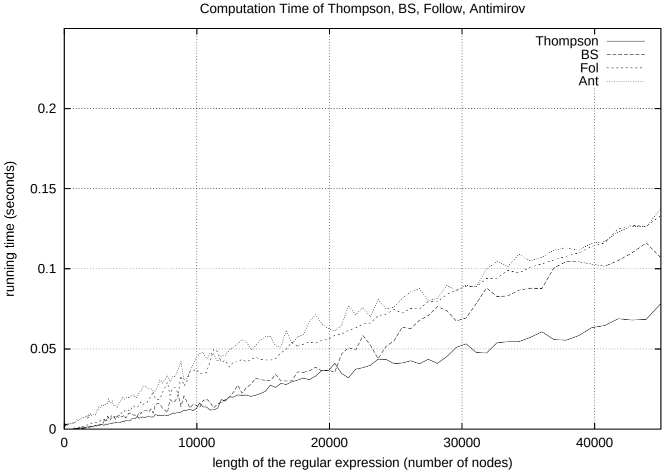
Equation automaton

$2, 4, 5 = 7, 9, 10$

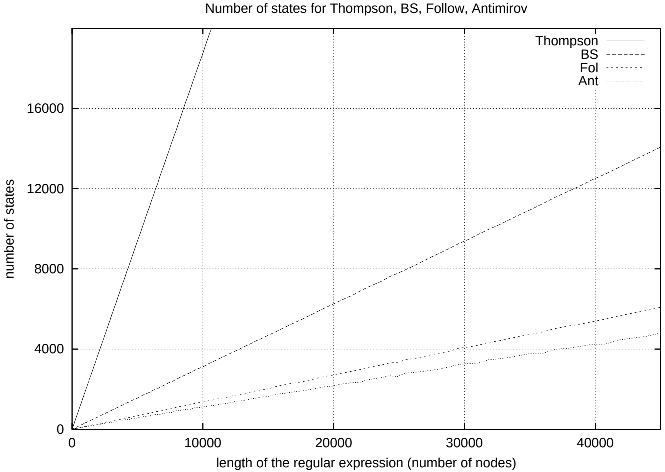


$3 = 8$

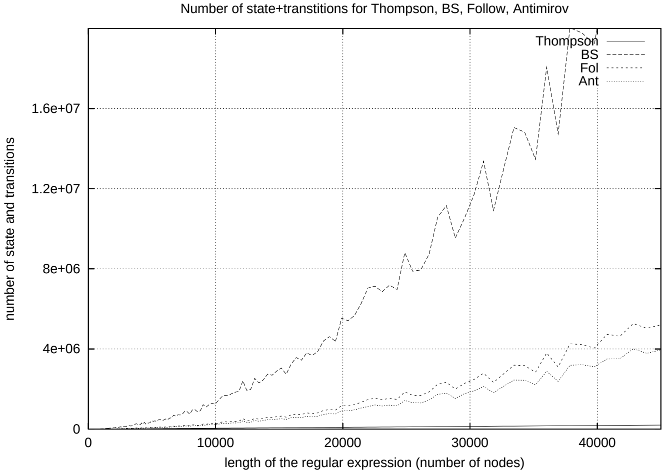
Benchmarks : time 1/3



Benchmarks : space (states) 2/3



Benchmarks : space (states+transitions) 3/3



Comparing algorithms

- Time:
  Thompson > Berry-Sethi > Follow > Antimirov

Comparing algorithms

- Time:
  Thompson > Berry-Sethi > Follow > Antimirov
- Space (states + transitions):
  Thompson > Antimirov > Follow > Berry-Sethi

Comparing algorithms

- Time:
  Thompson > Berry-Sethi > Follow > Antimirov
- Space (states + transitions):
  Thompson > Antimirov > Follow > Berry-Sethi
- Space (states):
  Antimirov > Follow > Berry-Sethi > Thompson

Comparing algorithms

- Time:
  Thompson > Berry-Sethi > Follow > Antimirov
- Space (states + transitions):
  Thompson > Antimirov > Follow > Berry-Sethi
- Space (states):
  Antimirov > Follow > Berry-Sethi > Thompson
- Implementation simplicity :
  Thompson > Berry-Sethi > Follow > Antimirov

Prove algorithms

Definition (Brzozowski's derivatives (1964))

$$a^{-1}a = 1$$
$$a^{-1}b = 0, \text{avec } b \neq a$$
$$a^{-1}(E + F) = a^{-1}E + a^{-1}F$$
$$a^{-1}(E \cdot F) = (a^{-1}E) \cdot F + \delta(E) \cdot (a^{-1}F)$$
$$a^{-1}(E^*) = (a^{-1}E) \cdot E^*$$

Definition (derivatives on words)

$$aw^{-1}(E) = w^{-1}(a^{-1}(E))$$

Prove algorithms

Using the following axioms

- $E \cdot \mathsf{1} = \mathsf{1} \cdot E = E$
  $E \cdot \mathsf{0} = \mathsf{0} \cdot E = \mathsf{0}$
  $E + \mathsf{0} = \mathsf{0} + E = E$
- ACI (Associative, commutative, idempotent)
  $(E + F) + G = E + (F + G)$
  $E + F = F + E$
  $E + E = E$

Theorem (Brzozowski 1964)

The set of derivatives is finite (modulo the above axioms).

Corollary (Brzozowski algorithm)

The set of derivatives are the states and the derivatives of derivatives are the transitions of a deterministic automaton.

Extension
Regular expressions – Rational expressions

We talk about Rational expressions when they are annotated with element of a semiring $\mathbb{K}$. This semiring is useful for dealing with.

- Multiplicities
- Weight
- ...

The algorithms presented may be extended for rational expressions.

Conclusion

- Eilenberg Machines offer a general model of non-deterministic computation, with a finite control and a computable relational data semantics.
- Simulation using a programming language with *polymorphism & higher-order constructions* (OCaml).
- The *reactive engine* is mathematically rigorous and a good methodology for simulating *Effective Eilenberg machines*.
- Compile regular expressions into automata efficiently in an applicative manner (OCaml).
- A new paradigm : Relational programming.

Thank You !

Locally finite relations

A finite subset of $\mathcal{D}$ enumerated by a finite stream:

type `stream` $\mathcal{D}$ `=`
  | `EOS`
  | `Stream` ***of*** $\mathcal{D}$ $\times$ `(delay` $\mathcal{D}$`)`

and `delay` $\mathcal{D}$ `= unit -> stream` $\mathcal{D}$`;`

Relations of $Rel(\mathcal{D})$ are **curryfied** and thus seen as functions thanks to the following isomorphism: $\wp(\mathcal{D} \times \mathcal{D}) = \mathcal{D} \rightarrow \wp(\mathcal{D})$

type `relation` $\mathcal{D}$ `=` $\mathcal{D}$ `-> stream` $\mathcal{D}$`;`

Finite Eilenberg Machines as a Functor

A machine $\mathcal{M} = (Q, \delta, I, T)$ on data $\mathcal{D}$ is the following module signature :

```
module type Machine = sig
  type 𝒟;
  type Q;
  value transition : Q -> list (relation 𝒟 × Q);
  value initial : list Q;
  value terminal : Q -> bool;
end;
```

We provide a **functor** :

```
module Engine (M : Machine) = sig
  value characteristic : relation 𝒟 ;
end;
```

The Reactive Engine in ML

```
(* react: D -> Q -> 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 -> Q -> 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 Reactive Engine in **Coq**

```
Program Fixpoint react (d : data) (s : state) (res : resumption)
  (h1 : WellFormedRes res)
  (h : Acc Rext ((Chi (d, s) (S (length (transition s))) O) :: (chi_res res)))
  {struct h} : (stream data) :=
  if terminal s
  then Stream data d (fun x:unit ⇒choose d s (transition s) res h1 _ _)
  else choose d s (transition s) res h1 _ _
with choose (d : data) (s : state) (ch : choice) (res : resumption)
  (h1 : WellFormedRes res) (h2 : incl ch (transition s))
  (h : Acc Rext ((Chi (d, s) (length ch) O) :: (chi_res res)))
  {struct h} : (stream data) :=
  match ch with
  | [] ⇒continue res h1 _
  | (rel, s') :: rest ⇒
    match (rel d) with
    | EOS ⇒choose d s rest res h1 _ _
    | Stream d' del ⇒react d' s' ((Choose d s rest rel del s') :: res) _ _
    end
  end
with continue (res : resumption) (h1 : WellFormedRes res)
  (h : Acc Rext (chi_res res)) {struct h} : (stream data) :=
  match res with
  | [] ⇒EOS data
  | back :: res' ⇒
    match back with
    | Advance d s ⇒react d s res' _ _
    | Choose d s rest rel del s' ⇒
      match (del tt) with
      | EOS ⇒choose d s rest res' _ _ _
      | Stream d' del' ⇒react d' s' ((Choose d s rest rel del' s') :: res') _ _
      end
    end
  end.
```

Engine *vs* Machine

We make a distinction between the terminology **"engine"** and **"machine"**. A machine can be non-deterministic whereas an engine is a deterministic process able to simulate a non-deterministic one. Finite Eilenberg machines describe non-deterministic computations which are enumerated by a deterministic process: the reactive engine.

Reminder: tries

We recall the structure of lexical trees or tries. A lexicon uses tries to store words letter by letter. Common initial substrings are shared. Nodes are marked with a boolean indicating membership.
Tries may be seen as deterministic finite state automata recognizing finite languages. Furthermore their sharing as dags yields the corresponding minimal fsa.
More generally, finite state automata state spaces may be represented as annotated tries, where the skeleton trie serves to address the states, and non-deterministic transitions are annotations, cycles being encoded by virtual adresses. This way, general finite-state machines may be represented applicatively, and minimized as dags.

References

- Sanskrit site: `http://pauillac.inria.fr/~huet/SKT/`
- Sandhi Analysis paper: `http://pauillac.inria.fr/~huet/FREE/tagger.ps`
- Course notes: `http://pauillac.inria.fr/~huet/ZEN/esslli.ps`
- Course slides: `http://pauillac.inria.fr/~huet/ZEN/Trento.ps`
- ZEN library: `http://pauillac.inria.fr/~huet/ZEN/zen.tar`
- Objective Caml: `http://caml.inria.fr/ocaml/`