Simulating Finite Eilenberg Machines with a Reactive Engine

Benoît Razet

INRIA Paris-Rocquencourt

MSFP 2008
Reykjavík, Iceland
2008 July 6th
The Eilenberg Machines Model
Built on Finite Automata
Eilenberg Machines

Built on Finite Automata

Let $S = (S, \cdot, 1)$ be a monoid, An $S$-automaton $A = (Q, \delta, I, T)$ : $Q$ finite set, $\delta$ 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** : $\text{lbl}(p) = s_1 \cdot \cdots \cdot s_n$
- **valid path** : $vp(A)$, $q_0 \in I$ et $q_n \in T$
- The **behavior** of the automaton is the set of all labels of valid paths : $|A| = \{ \text{lbl}(p) \mid p \in vp(A) \}$. 

Remark : $S$ is a monoid with no restriction (not necessarily free). If $S$ is instantiated with $\Sigma^*$ then the behavior defines a **language**, or else with $\Sigma^* \times \Gamma^*$ then it is a **relation on words**.
Let \( S = (S, \cdot, 1) \) be a monoid, An \( S \)-automaton \( A = (Q, \delta, I, T) \) :
\( Q \) finite set, \( \delta \) function \( Q \to \mathcal{P}(S \times Q) \), \( I \subseteq Q \), \( T \subseteq Q \).
One defines :

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

**Remark** : \( S \) is a monoid with no restriction (not necessarily free). If \( S \) is instantiated with \( \Sigma^* \) then the behavior defines a **language**, or else with \( \Sigma^* \times \Gamma^* \) then it is a **relation on words**.
A Relational Model

Let $\mathcal{D}$ be an abstract set, for the data. A relation $\rho$ 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.

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

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

Let $\mathcal{D}$ be an abstract set, for the data. A relation $\rho$ 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.

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

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

Finite Eilenberg Machines

The Reactive Engine

Example
Eilenberg Machines

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

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

\( \mathcal{D} \) is an abstract set, for the data.
An Eilenberg Machine is a \( \text{Rel}(\mathcal{D}) \)-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** : \( \text{lbl}(p) = \rho_1 \circ \cdots \circ \rho_n \)
- **valid path** : \( \text{vp}(\mathcal{M}), q_0 \in I \text{ et } q_n \in T \)
- **behavior** : \( |\mathcal{M}| = \{\text{lbl}(p) \mid p \in \text{vp}(\mathcal{M})\} \)
**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** : \( \text{lbl}(p) = \rho_1 \circ \cdots \circ \rho_n \)
- **valid path** : \( \text{vp}(\mathcal{M}), q_0 \in I \text{ et } q_n \in T \)
- **behavior** : \( |\mathcal{M}| = \{ \text{lbl}(p) \mid p \in \text{vp}(\mathcal{M}) \} \)

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

\[ ||\mathcal{M}|| = \bigcup_{\rho \in |\mathcal{M}|} \rho \]
Let $\mathcal{M}_i$ be a family of machines, consider the family of characteristic relations $||\mathcal{M}_i||$. This family can used as labels for another machine:
Automata, transducers, pushdown automata and Turing machines

Automata, rational transducers, pushdown automata and Turing machines have in common a finite state control which 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:

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

Samuel Eilenberg, Marcel-Paul Schützenberger, Seymour Gingsburg (ICALP 1972 at IRIA)
Finite Eilenberg Machines
Locally finite relations

A finite subset as a finite stream:

**Inductive** stream (data: Set) : Set :=

| EOS : stream data
| Stream : data →(unit →stream data) →stream data.

**Definition** delay (data: Set) : Set := unit →stream data.

Locally finite relations as functions using currying:

**Definition** relation (data: Set) : Set := data →stream data.

This is the *first* restriction on Eilenberg machines.
There is a membership predicate In_stream similar to the predicate In in library List.
Eilenberg Machines with locally finite relations

A machine is an automaton \((X, Q, \delta, I, T)\):

**Module Type** Machine.
- **Parameter** data: Set.
- **Parameter** state: Set.
- **Parameter** transition: state \(\rightarrow\) list ((relation data) * state).
- **Parameter** initial: list state.
- **Parameter** terminal: state \(\rightarrow\) bool.

*End Machine.*

We define \(\text{edge} : (d, q) \xrightarrow{\rho} (d', q')\) with \((\rho, q') \in \delta(q)\) and \(d' \in \rho(d)\).

**Definition** edge d q rel d' q' : Prop :=

\[
\text{In (rel,q') (transition q) } \land \text{In_stream d' (rel d).}
\]

**Definition** cell : Set := data * state.
We use the well-founded library of Coq with its \textit{accessibility} predicate:

\textbf{Inductive} \textit{Acc} \ (R \ : \ A \to A \to Prop) \ (x: A) \ : \ Prop \ := \\
\quad \| \text{Acc_intro} : (\forall y:A, R \ y \ x \to \text{Acc} \ R \ y) \to \text{Acc} \ R \ x.  \\

\textbf{Definition} \ \textit{well_founded} \ (R : A \to A \to Prop) \ := \ \forall a:A, \text{Acc} \ R \ a.  \\

\textbf{Definition} \ \textit{Rcell} \ (c' c : cell) \ : \ Prop \ := \\
\quad \exists \text{rel : relation data, edge} \ (\text{fst} \ c) \ (\text{snd} \ c) \ \text{rel} \ (\text{fst} \ c') \ (\text{snd} \ c').  \\

\textbf{Hypothesis} \ \textit{WfRcell} : \ \textit{well_founded} \ \textit{Rcell}.  \\

\textbf{The nötherian condition}
Solutions of the machine, and the simulation Theorem

Let $d_0$ be a data.

$d_n$ is a **solution** if and only if there exists a path of edges in the machine:

$$(d_0, q_0) \xrightarrow{\rho_1} (d_1, q_1) \xrightarrow{\rho_2} \ldots \xrightarrow{\rho_n} (d_n, q_n)$$

with $q_0$ an initial state and $q_n$ a terminal state.

**Theorem** simulation:

$\exists f : (\text{relation \ data}),$

$\forall (d \ d' : \text{data}), \text{Solution } d \ d' \iff \text{In\_stream } d' (f \ d).$
The Reactive Engine
The Reactive Engine Using Coq’s Program Extension

Program Fixpoint react (d : data) (s : state) (res : resumption)
    (h1 : WellFormedRes res)
  (h : Acc Rext ((Chi (d, s) (S (length (transition s))) 0) :: (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) 0) :: (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.

The reactive engine

- Enumeration of all solutions in a *depth first search* manner.
- Tail recursive algorithm.
- A reactive process defined with three mutually recursive functions.
- An extra parameter in the recursion: a stack for **backtracking**.
- Termination using a well-founded predicate as an extra parameter.
- Correctness.

**Remark. Engine versus 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.
A well-founded relation for the termination

Let $\mathcal{D}$ be a well-founded set by relation $<$, we define a relation $\prec$ on lists of $\mathcal{D}$. That is

$$l \prec d :: l$$

$$d_1 < d_2 \quad \Rightarrow \quad d_1 :: l \prec d_2 :: l$$

$$d_1 < d_3 \wedge d_2 < d_3 \quad \Rightarrow \quad d_1 :: d_2 :: l \prec d_3 :: l$$

Theorem WfRext : well_founded $<$ $\rightarrow$ well_founded $\prec$.

For instance

$$[3] \succ [2; 2] \succ [1; 1; 2] \succ [0; 1; 2] \succ [1; 2] \succ [0; 0; 2] \succ [0; 2] \succ [2] \cdots$$

This is a subcase of the multiset-ordering originally used in the reactive engine specification.
Correctness of the reactive engine

Before running the machine on data $d$, one must prepare all possible entry points in the automaton:

**Definition** $\text{init_res} (d : \text{data}) (l : \text{list state}) := \text{List.map (fun s ⇒Advance d q)} l.$

Then we call the function `continue` of the reactive engine on this initialization:

**Program Definition** $\text{characteristic_relation : relation data} := \text{fun (d: data) ⇒continue (init_res d initial)} _ _.$

**Theorem** $\text{correctness : } \forall d d', \quad \text{Solution } d \ d' \Longleftrightarrow \text{In_stream } d' (\text{characteristic_relation } d).$
A complete backtracking parser for $\lambda$-calculus
Consider the following \(\lambda\)-calculus ambiguous grammar:

\[
T \ ::= \ x \quad \text{(variable)} \\
     | \ \lambda x \cdot T \quad \text{(abstraction)} \\
     | \ T @ T \quad \text{(application)} \\
     | \ (T)
\]

Following this grammar the \(\lambda\)-term "\(\lambda x.x @ \lambda x.x\)" may be recognized as "\(\lambda x.(x @ \lambda x.x)\)" but also as "\((\lambda x.x) @ (\lambda x.x)\)".
A pushdown automaton for $\lambda$-calculus
Benchmarks

Finding all Solutions:

- \( \lambda x. x \times (\lambda x. x \times x) \times x \times x \times \lambda x. x \times x \) : 522 solutions instantaneously.
- \( x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \times x \) : 208012 solutions : 9 seconds of running time.
Conclusion

- Formal specification of a non-deterministic computational model and its simulation.
- Finite Eilenberg machines are specified and simulated using the reactive engine in Coq; mathematically rigorous.
- Not limited to the rational level, context-free example...
- The extraction technology of Coq is effective.
- Reusability: our technique is available for any problem presented as a finite Eilenberg machine. We provide a certified simulator.
- It will appear in the Zen toolkit.

I thank Gérard Huet, Matthieu Sozeau, Jean-Baptiste Tristan and Nicolas Pouillard.
Conclusion

- Formal specification of a non-deterministic computational model and its simulation.
- Finite Eilenberg machines are specified and simulated using the reactive engine in Coq; mathematically rigorous.
- Not limited to the rational level, context-free example...
- The extraction technology of Coq is effective.
- Reusability: our technique is available for any problem presented as a finite Eilenberg machine. We provide a certified simulator.
- It will appear in the Zen toolkit.

I thank Gérard Huet, Matthieu Sozeau, Jean-Baptiste Tristan and Nicolas Pouillard.
Thank you!
References

- Benoît Razet, *Finite Eilenberg Machines*, *CIAA* 2008
- Benoît Razet, *Simulating Finite Eilenberg Machines with a Reactive Engine*, *MSFP* 2008
- The Zen Toolkit, sanskrit.inria.fr/ZEN
The first solution

For randomly generated ambiguous $\lambda$-terms:
All solutions

For randomly generated *unambiguous* $\lambda$-terms (with all parentheses):

![Graph showing running time vs. length of words](image-url)
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)**

- For all relation $\rho$ labelling $\mathcal{M}$, $\rho$ is a locally finite relation: for all data $d$, the set $\rho(d)$ is finite.
- The length of any path necessarily is finite.

**Proposition (Koenig’s Lemma)**

The characteristic relation $||\mathcal{M}||$ is a **locally finite relation**.
There are two cases for which the second 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 well-founded relation $\rho$ on $D$ such that for all relation $\rho$ of the machine, for all data $d$ and $d'$,

$$d' \in \rho(d) \Rightarrow d > d'$$
Discussion about automata

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

- Deterministic Automata (DFA) OK.
- Non-deterministic Automata (NFA) OK.
- Automata with $\epsilon$-transitions It depends.
Discussion about automata

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

- Deterministic Automata (DFA) OK.
- Non-deterministic Automata (NFA) OK.
- Automata with ε-transitions It depends.

without ε-cycle OK.
The Word problem

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

$$
\begin{align*}
\mathcal{Q} = & \{q_1, q_2, q_3, q_4\}, \\
\Sigma = & \{a, b, c, d\}, \\
\mathcal{R} = & \{q_1 \rightarrow q_2, q_2 \rightarrow q_3, q_3 \rightarrow q_4, q_4 \rightarrow q_1\}.
\end{align*}
$$

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

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

- Based on Gérard Huet specification of the reactive engine in ZEN.
- The two restrictions are crucial, because it allows use to prove the termination of the reactive engine and then define it as a usual Coq function.
- Restriction of the multiset-ordering.
- Dependent types are useful for defining functions with invariants.
- Beautiful presentation using the PROGRAM extension of Coq by Matthieu Sozeau (Coq team, LRI Orsay).
- Extraction works pretty well ⇒ OCaml program.
Discussion

The simulation of non-deterministic computational model using a deterministic process needs generally a breadth first search for solutions enumeration. The finite Eilenberg machines model only requires a depth first search strategy.

It has complexity advantages (*memory and time*).
PROGRAM extension

Using the PROGRAM extension of Coq we obtain a readable program definition. It is due to two features of PROGRAM. First, it is allowed to give function definitions without providing all logical justifications which are delayed as proof obligations. Each underscore character in the body of the function creates a proof obligation according to function declarations. Secondly, PROGRAM brings the following enhancement concerning the dependent pattern matching:

```
match v return T with
| v1 ⇒ t1
| ... ⇒ ...
| vn ⇒ tn
end
```

```
match v as x return v = x → T with
| v1 ⇒ (fun eq ⇒ t1)
⇒ | ... ⇒ ...
| vn ⇒ (fun eq ⇒ tn)
end (refl_equal v)
```