Computing with Relational Machines

Gérard Huet and Benoît Razet

INRIA Paris-Rocquencourt

In memory of Peter Landin

Abstract. We propose a relational computing paradigm based on Eilenberg machines, an effective version of Eilenberg's X-machines suitable for general non-deterministic computation. An Eilenberg machine generalises a finite-state automaton, seen as its control component, with a computation component over a data domain specified as an action algebra. Actions are interpreted as binary relations over the data domain. We show various strategies for the sequential simulation of our relational machines, using variants of the *reactive engine*. In a particular case of *finite machines*, we show that bottom-up search yields an efficient complete simulator.
Relational machines may be composed in a modular fashion, since atomic actions of one machine can be mapped to the characteristic relation of other relational machines acting as its parameters.
The control components of machines can be compiled from regular expressions. Several such translations have been proposed in the literature, which we briefly survey.
Our view of machines is completely applicative. They can be defined constructively in type theory, where the correctness of their simulation may be formally checked. From formal proofs in the Coq proof assistant, efficient functional programs in the Objective Caml programming language can be mechanically extracted.
Most of this material is extracted from the doctoral research of the second author[21]. A preliminary version of this paper was presented as a tutorial at ICON'2008 in Pune, Maharashtra, in December 2008.

1 Introduction

Peter Landin deserves special credit for the effective dissemination of functional programming. His landmark paper "The next 700 programming languages"[18] introduces a crucial extension ISWIM of pure lambda-calculus for use as a programming language, firstly by proposing the *let* notation for explicit redexes, and secondly by adding primitive control operators for conditional and recursion, and primitive data structures such as booleans and integers for direct higher-order recursions. ISWIM was complemented by a polymorphic type system and the operators of cartesian product by Robin Milner, yielding a very clean functional language, ML, fit for deterministic programming.

Computing in a non-deterministic way has not been adapted to a uniform elegant programming language in the same manner so far. Of course, Dijkstra's Guarded Command Language (GCL) allows non-deterministic choice in pattern matching, but it stayed as a pencil and paper language usable for proofs in predicate transformer semantics, rather than production programming. Prolog is an attempt at logic programming seen as the non-deterministic search for the satisfiability of Horn clauses, but it is notoriously weak at controling the search for solutions, in the absence of programmable tactics. The Prolog user can control this search solely by reordering the clauses of the program, and inserting "cut" primitives that break the applicative semantics of the language. Later constraint programming languages suffer similar problems, and the user of such "black-box computers" is torn between his fear of ad-hoc programming of backtracking processes in a conventional deterministic language, and his loathing of being the hostage of generic tactics ill-fitted to his particular problem.

Actually non-deterministic computation is fairly straightforward if one has a clear notion of the search space of the problem, if fairness is enforced (either by termination or by non-starvation tactics), and if higher-order applicative programming is used. Higher-order gives you continuations and streams for your enumerative processes, and when you resume search at backtracking points you retrieve the stored state of the computing thread, without risk of corruption by side effects. This style of programming may be systematized within a very general framework for relational computation, using ML as its core deterministic engine. Furthermore, the control of such non-deterministic relational search fits well within a very elegant proposal of Samuel Eilenberg for generalizing finite-state automata with more abstract X-machines[9]. Formal semantics of computable relations may then be developed within the *actions* algebras of Vaughan Pratt[19].

We propose here this relational programming paradigm, implemented as an ML library of parametric modules, as a tribute to Peter Landin and Samuel Eilenberg.

2 Machines

2.1 Relational machines

We shall define a notion of abstract machine inspired from the work of Eilenberg (X-machines, presented in his *magnum opus* on automata theory[9]). Our machines are non-deterministic in nature. They comprise a *control component*, similar to the transitions state diagram of a (non-deterministic) automaton. These transitions are labeled by action generators. Action expressions over free generators, generalizing regular expressions from the theory of languages, provide a specification language for the control component of machines. A program, or action expression, compiles into control components according to various translations. Control components in their turn may be compiled further into transition matrices or other representations.

Our machines also comprise a *data component*, endowed with a relational semantics. That is, we interpret action generators by semantic attachments to binary relations over the data domain. These relations are themselves represented as functions from data elements to streams of data elements. This applicative apparatus replaces the imperative components of traditional automata (tapes, reading head, counters, stacks, etc) by clear mathematical notions.

We shall now formalise these notions in a way that will exhibit the symmetry between control and data. First of all, we postulate a finite set Σ of parameters standing for the names of the primitive operations of the machine, called *generators*.

For the control component, we postulate a finite set S of states and a *transition relation map* interpreting each generator as a (binary) relation over S. This transition relation interpretation is usually presented as a curried *transition function* δ mapping each state in S to a finite set of pairs (a, q) with a a generator and q a state. This set can in turn be implemented as a finite list of such pairs.

Finally, we select in S a set of *initial states* and a set of *accepting states*.

For the data component, we postulate a set D of data values and a *computation relation map* interpreting each generator as a (binary) relation over D. Similarly as for the control component, we shall curry this relation map as a *computation function* mapping each generator a in Σ to a function $\rho(a)$ in $D \to \wp(D)$. Now the situation is different from control, since D and thus $\wp(D)$ may be infinite. In order to have a constructive characterization, we shall assume that D is recursively enumerable, and that each $\rho(a)$ maps $d \in D$ to a recursively enumerable subset of $\wp(D)$. We shall represent such subsets as progressively computed streams of values, as we shall explain in due time.

2.2 Progressive relations as streams

We recall that a recursively enumerable subset of ω is the range of a partial recursive function in $\omega \rightharpoonup \omega$, or equivalently it is either empty or the range of a (total) recursive function in $\omega \to \omega$. None of these two definitions is totally satisfying, since in the first definition we may loop on some values of the parameter, obliging us to dovetail the computations in order to obtain a sequence of elements that completely enumerates the set, and in the second we may stutter enumerating the same element in multiple ways. This stuttering cannot be totally eliminated without looping, for instance for finite sets. Furthermore, demanding total functions is a bit illusory. It means either we restrict ourserves to a non Turing-complete algorithmic description language (such as primitive recursive programs), or else we cannot decide the totality of algorithms demanded by the definition.

We shall here assume that our algorithmic description language is ML, in other words, typed lambda-calculus evaluated in call by value with a recursion operator, inductive types and parametric modules. More precisely, we shall present all our algorithms in Pidgin ML, actually the so-called “revised syntax” of Objective Caml.

In this framework we can define computable streams over a parametric datatype `data` as follows:

```
type stream 'data =
 [ Void
 | Stream of 'data and delay 'data
 ]
and delay 'data = unit → stream 'data;
```

This definition expresses that a stream of data values is either `Void`, representing the empty set, or else a pair `Stream(d,f)` with d of type `data`, and f a frozen stream value, representing the set $\{d\} \cup F$, where F may be computed as the stream $f()$, where $()$ is syntax for the canonical element in type `unit`. Using this inductive parametric datatype, we may now define progressive relations by the following type:

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

2.3 Kernel machines

We now have all the ingredients to define the module signature of *kernel machines*:

```
module type EMK = sig
 type generator;
 type data;
 type state;
 value transition: state → list (generator × state);
 value initial: list state;
 value accept: state → bool;
 value semantics : generator → relation data;
end;
```

In the following, we shall continue to use Σ (resp. D, S, δ, ρ) as shorthand for `generator` (resp. `date`, `state`, `transition`, `semantics`). We also write I for `initial` and T for the set of accepting states (for which the predicate `accept` is true).

A machine is like a blackbox, which evolves through series of non-deterministic computation steps. At any point of the computation, its status is characterized by the pair (s, d) of its *current state* $s \in S$ and its *current data value* $d \in D$. Such a pair is called a *cell.*

A computation step issued from cell (s, d) consists in choosing a transition $(a, s') \in \delta(s)$ and a value $d' \in \rho(a)(d)$. If any of these choices fails, because the corresponding set is empty, the machine is said to be *blocked*; otherwise, the computation step succeeds, and the machine has as status the new cell (s', d'). We write $(s, d) \xrightarrow{a} (s', d')$.

A *computation path* is a finite sequence of such computations steps:

$$(s_0, d_0) \xrightarrow{a_1} (s_1, d_1) \xrightarrow{a_2} (s_2, d_2) ... \xrightarrow{a_n} (s_n, d_n)$$

The computation is said to be *accepting* whenever $s_0 \in I$ and $s_n \in T$, in which case we say that the machine *accepts* input d_0 and *computes* output d_n. Note that (d_0, d_n) belongs to the graph of the composition of relations labeling the path: $\rho(a_1) \circ \rho(a_2) \circ ...\rho(a_n)$.

We have thus a very general model of relational calculus. Our machines compute relations over the data domain D, and we shall thus speak of *D-machines*. The "machine language" has the action generators for instructions. Actions compose by computation. Furthermore, a high-level programming language for relational calculus may be designed as an action calculus. The obvious point of departure for this calculus is to consider regular expressions, in other words the free Kleene algebra generated by the set of generators. We know from automata theory various translations from regular expressions to finite-state automata. Every such translation gives us a compiler of our action algebra into the control components of our machines: S, δ, I and T. The data components, D and ρ, offer a clean mathematical abstraction over the imperative paraphernalia of classical automata: reading heads, tapes, etc. And we get immediately a programming language enriching the machine language of primitive actions by composition, iteration, and choice.

Indeed, a finite automaton over alphabet Σ is readily emulated by the machine with generator set Σ having its state transition graph as its control component, and admitting the free monoid of actions Σ^* for data domain. Each generator a is interpreted in the semantics as the (functional) relation $\rho(a) = L_a^{-1} =_{def} \{(a \cdot w, w) \mid w \in \Sigma^*\}$ which "reads the input tape". And indeed the language recognized by the automaton is retrieved as the composition of actions along all accepting computations. Here the data computation is merely a trace of the different states of the "input tape".

This example is a particularly simple one, in which data computation is deterministic, since in this case $\rho(a)$ is a partial function. We may say that such a machine is "data driven". Control will be deterministic too, provided the underlying automaton is deterministic, since every $\delta(s)$ will then have a unique non-blocking transition. But note that the same control component could be associated with different semantics. For instance, with $\rho(a) = R_a =_{def} \{(w, w \cdot a) \mid w \in \Gamma^*\}$, the machine will enumerate with its accepting computations the regular language recognized by the automaton.

Let us now turn towards the action calculus.

3 Actions

Actions may be composed. We write $A \cdot B$ for the composition of actions A and B. This operator corresponds to the composition of the underlying relations.

Actions may be iterated. We write A^+ for the iteration of action A. This operator corresponds to the transitive closure of the underlying relation. We postulate an identity action 1 corresponding to the underlying identity relation.

Actions may be summed. We write $A + B$ for the sum of actions A and B. This corresponds to the union of the underlying relations. We note A^* for $1+A^+$. We also postulate an empty action 0.

The algebraic structure of actions is that of a composition monoid:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

$$A \cdot 1 = 1 \cdot A = A$$

It is completed, for union, as an idempotent abelian monoid:

$$(A + B) + C = A + (B + C)$$

$$A + B = B + A$$

$$A + 0 = 0 + A = A$$

$$A + A = A$$

We postulate distributivity of these two operations:

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

$$(A + B) \cdot C = A \cdot C + B \cdot C$$

$$A \cdot 0 = 0 \cdot A = 0$$

Thus, so far, actions form an idempotent semiring. Defining $A \leq B =_{def} A+B = B$, the partial ordering $\leq$ makes the algebra of actions an upper semilattice.

As for iteration (which will be interpreted over relations by transitive-reflexive closure), we follow Pratt [19] in adding implications between actions, in order to get an algebraic variety. This is in contradistinction with Kleene algebras, which only form a quasi variety, and need conditional identities for their complete axiomatisation as shown by Kozen[16]. Nevertheless, actions algebras form a conservative extension of Kleene algebras, and thus do not introduce spurious consequences for regular expressions.

Thus we postulate $\leftarrow$ and $\rightarrow$, corresponding to relational semi-complements:

$$\rho \rightarrow \sigma = \{(v, w) \mid \forall u \; u\rho v \Rightarrow u\sigma w\}$$

$$\sigma \leftarrow \rho = \{(u, w) \mid \forall v \; w\rho v \Rightarrow u\sigma v\}$$

and we axiomatise actions as *residuation algebras*, following Kozen [17]:

$$A \cdot C \leq B \Leftrightarrow C \leq A \rightarrow B$$

$$C \cdot A \leq B \Leftrightarrow C \leq B \leftarrow A$$

or alternatively we may replace these two equivalences by the following equational axioms:

$$A \cdot (A \rightarrow B) \leq B$$

$$(B \leftarrow A) \cdot A \leq B$$

$$A \rightarrow B \leq A \rightarrow (B + C)$$

$$B \leftarrow A \leq (B + C) \leftarrow A$$

$$A \leq B \rightarrow (B \cdot A)$$

$$A \leq (A \cdot B) \leftarrow B$$

We may now get Pratt's action algebras by axiomatizing iteration as 'pure induction':

$$1 + A + A^* \cdot A^* \leq A^*$$

$$(A \rightarrow A)^* = A \rightarrow A$$

$$(A \leftarrow A)^* = A \leftarrow A$$

The residuation/implication operations may be seen as the right interpolants to extend conservatively Kleene algebras to the variety of action algebras.

Furthermore, following Kozen [17], we may wish to enrich our actions with a multiplicative operation $\cap$, corresponding to relation intersection, verifying lower semilattice axioms:

$$(A \cap B) \cap C = A \cap (B \cap C)$$

$$A \cap B = B \cap A$$

$$A \cap A = A$$

We may also complete to a lattice structure with:

$$A + (A \cap B) = A$$

$$A \cap (A + B) = A$$

Thus we obtain Kozen's *action lattices*, the right structure for matrix computation.

We remark that such structures go in the direction of logical languages, since union, intersection and residuation laws are valid Heyting algebras axioms. Further extensions of interest to be envisioned are the inverse $\widetilde{A}$ (corresponding in the regular languages model to string reversal), the complement A^-, and the pair of derivatives $A \setminus B$ and $A \,/\, B$ (also called quotients). Residuals and derivatives are related through the identities $A \rightarrow B = (A \setminus B^-)^-$ and symmetrically $B \leftarrow A = (B^- \,/\, A)^-$. We remark that we are still far from the complete Boolean algebra structure of relations, though.

4 Behaviour, modularity, and interfaces

We recall that we defined above the accepting computations of a machine, and for each such computation its compound action, obtained by composing the generating relations of each computation step. Let us call *behaviour* of a machine $\mathcal{M}$ the set of all such compound actions, noted $|\mathcal{M}|$.

Now we define the *characteristic relation* of a machine $\mathcal{M}$ as the union of the semantics of its behaviour:

$$||\mathcal{M}|| = \bigcup_{a \in |\mathcal{M}|} \rho(a)$$

Characteristic relations are the relational interpretation over the data domain D of the action langage recognized by the underlying automaton. They allow us to compose our machines in modular fashion.

4.1 Modular construction of machines

Now that we understand that a D-machine implements a relation over D, we may compose machines vertically as follows. Let $\mathcal{A}$ be a (non-deterministic) automaton over alphabet Σ, and for every $a \in \Sigma$ let $\mathcal{N}_a$ be a D-machine over some generator set Σ_a. We may now turn $\mathcal{A}$ into a D-machine over generator set Σ by taking $\mathcal{A}$ as its control component, and extending it by a data component having as semantics the function mapping $a \in \Sigma$ to $||\mathcal{N}_a||$.

We may thus construct large machines from smaller ones computing on the same data domain. A typical example of application for computational linguistics is to do morphological treatment (such as segmentation and tagging of some corpus) in a lexicon-directed way. The alphabet Σ defines the lexical categories or parts of speech, each machine $\mathcal{N}_a$ implements access to the lexicon of category a, the automaton $\mathcal{A}$ defines the morphological geometry, and the composite machine $\mathcal{M}$ implements a lexicon-directed parts-of-speech tagger. By appropriate extension of the lexicon machines $\mathcal{N}_a$, morpho-phonemic treatment at the junction of the words may be effected, such as complete sandhi analysis for Sanskrit. This was the motivating example for which the Zen toolkit was designed [13, 14].

4.2 Interfaces

What we described so far is the Eilenberg machine *kernel*, consisting of its control and data elements. We may complete this description by an *interface*, composed of an input domain D_-, an output domain D_+, an input relation $\phi_- : D_- \rightarrow D$ and an output relation $\phi_+ : D \rightarrow D_+$. A kernel machine $\mathcal{M}$ completed by this interface I defines a relation $\phi(M, I) : D_- \rightarrow D_+$ by composition:

$$\phi(\mathcal{M}, I) = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$$

5 Finite machines

We shall now present an important special case of machines which exhibit a finite behaviour.

The relation $\rho : D \to D'$ is said to be *locally finite* if for every $d \in D$ the set $\rho(d)$ of elements of D related to d is finite. The machine $\mathcal{M}$ is said to be *locally finite* if every generating relation $\rho(a)$ is locally finite [11]. The machine $\mathcal{M}$ is said to be *nœtherian* if all its computations are finite in length.

We remark that a machine is nœtherian when its data domain D is a well-founded ordering for the order relation $>$ generated by:

$$d > d' \;\Leftarrow\; \exists a \in \Sigma \; d' \in \rho(a)(d)$$

Indeed, if there existed an infinite computation, there would exist an infinite sub-sequence going through the same state. But the converse is not true, since a machine may terminate for reasons depending of its control.

Finally, we say that a machine is *finite* if it is locally finite and nœtherian.

We say that a machine kernel is *deterministic* [9] iff $|I| \leq 1$ and for each cell value (s, d) occurring in a computation there exists at most one computation transition issued from it, i.e. if $\delta(s)$ is a set of pairs $\{(\rho_1, s_1), (\rho_2, s_2), ...(\rho_n, s_n)\}$ such that for at most one $1 \leq k \leq n$ the set $\rho_k(d)$ is non empty, and if such k exists then $\rho_k(d)$ is a singleton. This condition demands that on one hand the transition relation of the underlying automaton be a partial function, that is the automaton must be deterministic, and on the other hand that the relations leading out of a state s be partial functions over the subset of D which is reachable by computations leading to s. We extend this property to a machine with interface by requiring that its input relation ϕ_- and its output relation ϕ_+ be partial functions. We remark that a deterministic machine may nevertheless generate several solutions, since a terminal cell is not necessarily blocking further computation. Under certain extra conditions, such a machine computes a partial function (see chapter X, section 8 of Eilenberg's treatise [9]).

5.1 Examples

Non deterministic finite automata Let us consider a non-deterministic automaton $\mathcal{A}$ with parameters (S, I, T, δ). We construct an Eilenberg machine $\mathcal{M}$ solving the word problem for the regular language $|\mathcal{A}|$ recognized by the automaton. $\mathcal{M}$ has Σ for generating set, and it takes $\mathcal{A}$ for its control component. For the data component, we take $D = \Sigma^*$, and the semantics is defined as $\rho(a) = L_a^{-1} =_{def} \{(a \cdot w, w) \mid w \in \Sigma^*\}$, as explained above.

We may check that $\rho(w) = 1$ iff $w \in |\mathcal{A}|$. It is easy to check that $\mathcal{M}$ is finite, since data decreases in length, and semantics is a partial function. When $\mathcal{A}$ is a deterministic automaton, $\mathcal{M}$ is a deterministic machine.

Another machine with the same control component may be defined to enumerate all the words in set $|\mathcal{A}|$. In general it will neither be finite, nor deterministic.

Rational transducers Let Σ and Γ be two finite alphabets. A transducer $A : \Sigma \Rightarrow \Gamma$ is similar to a (non-deterministic) automaton, whose transitions are labeled with pairs of words in $D = \Sigma^* \times \Gamma^*$. Let Ω be the (finite) set of labels occurring as labels of the transitions of $\mathcal{A}$. The transition graph of $\mathcal{A}$ may thus be considered as an ordinary non-deterministic automaton over generator alphabet Ω, and constitutes the control component of the machines we shall define to solve various transductions tasks.

We recall that a transducer "reads its input" on an input tape representing a word in Σ^* and "prints its output" on an output tape representing a word in Γ^*. On transition (w, w') it reads off w on the input tape, and if successful appends w' to its output tape. If by a succession of transitions starting from an initial state with input i and empty output it reaches an accepting state with empty input and output o, we say that (i, o) belongs to the *rational relation* in $\Sigma \Rightarrow \Gamma$ recognized by the transducer $\mathcal{A}$, which we shall write $|\mathcal{A}|$. We shall now solve various decision problems on $|\mathcal{A}|$ using machines which use $\mathcal{A}$ for control and D for data, but replace the tapes by various semantic functions:

1. Recognition. Given $(w, w') \in D$, decide whether $(w, w') \in |\mathcal{A}|$.
2. Synthesis. Given $w \in \Sigma^*$, compute its image $|\mathcal{A}|(w) \subset \Gamma^*$.
3. Analysis. Given $w \in \Gamma^*$, compute the inverse image $|\mathcal{A}^{-1}|(w) \subset \Sigma^*$.

Recognition. The semantics ρ is defined by $\rho(\sigma, \gamma) = L_\sigma^{-1} \times L_\gamma^{-1}$. Like for ordinary automata we obtain a finite machine, provided the transducer has no transition labeled (ϵ, ϵ), since at least one of the two lengths decreases. We choose as interface $D_- = \Sigma^* \times \Gamma^*$, $\phi_- = Id_{\Sigma^* \times \Gamma^*}$, $D_+ = 0, 1$, $\phi_+(w, w') = 1$ iff $w = w' = \epsilon$.

Synthesis. The semantics ρ is defined by $\rho(\sigma, \gamma) = L_\sigma^{-1} \times R_\gamma$, with $R_\gamma =_{def} \{(w, w \cdot \gamma) \mid w \in \Gamma^*\}$. We choose as interface $D_- = \Sigma^*$, $\phi_- = \{(w, (w, \epsilon)) \mid w \in \Sigma^*\}$, $D_+ = \Gamma^*$, $\phi_+ = \{(\epsilon, w'), w') \mid w' \in \Gamma^*\}$. We get $|\mathcal{A}| = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$. Such a machine is locally finite, since relations L_σ^{-1} and R_γ are partial functions. However, it may not be nœtherian, since there may exist transitions labeled with actions (ϵ, w). Actually the machine is nœtherian iff cycles of such transitions do not occur, i.e., iff the set $|\mathcal{A}|(w)$ is finite for every $w \in \Sigma^*$ (see Razet [20]).

Analysis. Symmetric to synthesis, replacing L_σ^{-1} by R_σ and R_γ by L_γ^{-1}.

Oracle machines Let D be an arbitrary set, and P an arbitrary predicate over D. We consider the relation ρ over D defined as the restriction of identity to the data elements verifying P: $\rho(d) = \{d\}$ if $P(d)$, $\rho(d) = \emptyset$ otherwise. We define in a canonical way the machine whose control component is the automaton $\mathcal{A}$ with two states $S = \{0, 1\}$, $I = \{0\}$ and $T = \{1\}$, and transition function δ defined by $\delta(0) = \{(\rho, 1)\}$ and $\delta(1) = \emptyset$. This machine is a deterministic finite machine, that decides in one computational step whether its input verifies P. Our restriction of finite Eilenberg machines to computable relations limits such oracles to recursive predicates, but of arbitrary complexity. More generally, our machines recursively enumerate arbitrary recursively enumerable sets, and are therefore Turing complete.

6 Reactive engine

We may simulate the computations of a finite Eilenberg machine by adapting the notion de *reactive engine* of the Zen library [12–14, 20]. The engine is a deterministic simulator of the non-deterministic machine.

6.1 The depth-first search reactive engine

We start with a simple depth-first search engine, appropriate for finite machines. We define the engine as an ML functor, that is a module taking as parameter a kernel machine.

```
module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* We stack backtrack choice points in a resumption *)
type backtrack =
 [ React of data and state
 | Choose of data and choice and delay data and state
 ]
and resumption = list backtrack;

(* The 3 internal loops of the reactive engine (using terminal calls) *)

(* react: data → state → resumption → stream data *)
value rec react d q res =
 let ch = transition q in
 (* We need to compute [choose d ch res] but first
    we deliver data [d] to the stream of solutions when [q] is accepting *)
 if accept q
   then Stream d (fun () → choose d ch res) (* Solution d found *)
   else choose d ch res

(* choose: data → choice → resumption → stream data *)
and choose d ch res =
 match ch with
 [ [] → resume res
 | [ (g, q’) :: rest ] → match semantics g d with
             [ Void → choose d rest res
             | Stream d’ del → react d’ q’ [ Choose d rest del q’ :: res ]
             ]
 ]

(* The scheduler which backtracks in depth-first exploration *)
(* resume: resumption → stream data *)
and resume res =
 match res with
 [ [] → Void
 | [ React d q :: rest ] → react d q rest
 | [ Choose d ch del q' :: rest ] →
   match del () with (* We thaw the delayed stream of solutions *)
   [ Void → choose d ch rest (* Finally we look for next pending choice *)
   | Stream d' del' → react d' q' [ Choose d ch del' q' :: rest ]
   ]
 ]
;

(* Simulating the characteristic relation: relation data *)
value simulation d =
 let rec init_res l acc =
  match l with
  [ [] → acc
  | [ q :: rest ] → init_res rest [ React d q :: acc ]
  ] in
 resume (init_res initial [])
;

end; (* module Engine *)
```

This reactive engine has a very simple management of pending choices, since the backtrack choices are stored on a resumption stack, last-in first-out. It is very fast, since the ML compiler replaces the terminal calls by jumps. It is the workhorse of the original motivating application, Sanskrit sentence segmentation [13].

6.2 Correctness, completeness, certification

A proof of correctness and completeness of this simulator was given by Huet in the case of segmentation transducers [13]. Benoît Razet generalized the proof to the general case of finite Eilenberg machines, and formalized it in the Coq proof assistant [22]. From the formal proof object it is possible to extract mechanically ML algorithms identical to the ones we showed above.

6.3 A general reactive engine, driven by a strategy

When a machine is not finite, and in particular when there are infinite computation paths, the above bottom-up engine may loop, and the simulation is not complete. In order to remedy this problem, we shall change the specific last-in first-out policy of resumption management, and replace it by a more general strategy, given as an extra parameter of the machine.

```
open Eilenberg;

module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* We separate the control choices and the data relation choices *)
type backtrack =
 [ React of data and state
 | Choose of data and choice
 | Relate of stream data and state
 ]
;
```

Now `resumption` is an abstract data type, given in a module `Resumption`, passed as argument to the `Strategy` functor, and generalizing a backtrack stack.

```
module Strategy (* resumption management *)
 (Resumption : sig
   type resumption;
   value empty: resumption;
   value pop: resumption → option (backtrack × resumption);
   value push: backtrack → resumption → resumption;
 end) =
 struct

open Resumption;
```

We now modify the reactive engine, so that resumption management is governed by the given strategy.

```
(* react: data → state → resumption → stream data *)
value rec react d q res =
 let ch = transition q in
 if accept q (* Solution d found? *)
   then Stream d (fun () → resume (push (Choose d ch) res))
   else resume (push (Choose d ch) res)

(* choose: data → choice → resumption → stream data *)
and choose d ch res =
 match ch with
 [ [] → resume res
 | [ (g, q') :: rest ] →
   let res' = push (Choose d rest) res in
   relate (semantics g d) q' res'
 ]
```

```
(* relate: stream data → state → resumption → stream data *)
and relate str q res =
 match str with
 [ Void → resume res
 | Stream d del →
   let str = del () in
   resume (push (React d q) (push (Relate str q) res))
 ]

(* resume: resumption → stream data *)
and resume res =
 match pop res with
 [ None → Void
 | Some (b, rest) →
   match b with
   [ React d q → react d q rest
   | Choose d ch → choose d ch rest
   | Relate str q → relate str q rest
   ]
 ]
;

(* characteristic_relation: relation data *)
value simulation d =
 let rec init_res l acc =
   match l with
   [ [] → acc
   | [ q :: rest ] → init_res rest (push (React d q) acc)
   ] in
 resume (init_res initial empty)
;

end; (* module Strategy *)
```

6.4 A few typical strategies

We now give a few variations on search strategies. First of all, we show how the original depth-first reactive engine may be obtained by a `DepthFirst` strategy module, adequate for finite Eilenberg machines.

```
module DepthFirst = struct
 type resumption = list backtrack;
 value empty = [];
 value push b res = [ b :: res ];
 value pop res =
   match res with
```

```
  [ [] → None
  | [ b :: rest ] → Some (b,rest)
  ];
end; (* module DepthFirst *)
```

For the record, here is a breadth-first strategy module, using FIFO queues for resumption management:

```
module BreadthFirst = struct
  type resumption = (list backtrack * list backtrack);
  value empty = ([],[]);
  value push b res =
    let (input, output) = res in
    ([ b :: input ], output)
  ;
  value pop res =
    let (input, output) = res in
    match output with
    [ [] → let new_output = List.rev input in
           match new_output with
           [ [] → None
           | [ b :: rest ] → Some (b, ([],rest))
           ]
    | [ b :: rest ] → Some (b, (input,rest))
    ]
  ;
end; (* module BreadthFirst *)
```

We remark that there is a cost involved in reversing the *input* list above, and that suppressing this reversing operation yields a more efficient machine working in a "boustrophedon" manner, while keeping a fairness property:

```
module Fair = struct
 type resumption = (list backtrack × list backtrack);
 value empty = ([],[]);
 value push b res =
   let (left,right) = res in
   (left, [ b :: right ])
 ;
 value pop res =
   let (left,right) = res in
   match left with
   [ [] → match right with
          [ [] → None
          | [ r :: rrest ] → Some (r, (rrest,[]))
          ]
   | [ l :: lrest ] → Some (l, (lrest,right))
   ]
 ;
end; (* module Fair *)
```

Finally, we examine the special case of deterministic machines. The following simple `Det` tactic is adapted to this case.

```
module Det = struct
 type resumption = list backtrack;
 value empty = [];
 value push b res =
   match b with
   [ React _ _ → [ b :: res ]
   | Choose _ _ → [ b ] (* cut : the list contains only one element *)
   | Relate _ _ → res (* no other delay *)
   ];
 value pop res =
   match res with
   [ [] → None
   | [ b :: rest ] → Some (b,rest)
   ];
end; (* module Det *)
```

Now we may build the various modules encapsulating the various strategies.

```
module FEM = Strategy DepthFirst; (* The bottom-up engine *)
module Fair_Engine = Strategy Fair; (* A fair engine *)
module Deterministic_Engine = Strategy Det; (* The deterministic one *)

end; (* module Engine *)
```

Nevertheless, a complete evaluation strategy in the general case demands a more complex stream definition, where the computation is sliced into provably terminating states. This extension is discussed in Razet's thesis [21].

7 From regular expressions to automata

Our motivation here is the design of a language for describing the control part of Eilenberg machines. The control part of Eilenberg machines is a finite automaton. It naturally leads us to *regular expressions* and their translations into finite automata.

There have been more than 50 years of research on the problem of compilation (or translation) of regular expressions into automata. It started with Kleene who stated the equivalence between the class of languages recognized by finite automata and the class of languages defined by regular expressions. This topic is particularly fruitful because it has applications to string-search algorithms, circuits, synchronous languages, computational linguistics, *etc.* This wide range of applications leads one to several automata and regular-expressions variants.

Usually, an algorithm compiling regular expressions into automata is described in an imperative programming style for managing states and edges: states are allocated, merged or removed and so on concerning the edges. However, and this may seem somewhat surprising, it is possible to describe each of the well-known algorithms in an applicative manner, while preserving its computational complexity. This methodology leads to formal definitions of the algorithms exhibiting important invariants, an essential step towards their formal verification.

We focus on fast translations, whose time complexity is linear or quadratic with respect to the size of the regular expression. First we present *Thompson*'s algorithm [23] and then we review other algorithms that are may be put to use by our methodology.

Let us mention Brzozowski's algorithm [5], which translates a regular expression (even with Boolean operators) into a *deterministic* automaton. Unfortunately, its complexity is theoretically exponential in space and time. Nevertheless, it introduced the notion of regular expression *derivative* which is a fundamental idea pervading other algorithms.

7.1 Thompson's algorithm

Thompson presented his algorithm in 1968 and it is one of the most famous translations. It computes a finite non-deterministic automaton with ϵ-moves in linear time.

Let us first define regular expressions as the following datatype:

```
type regexp 'a =
  [ One
  | Symb of 'a
  | Union of regexp 'a and regexp 'a
  | Conc of regexp 'a and regexp 'a
  | Star of regexp 'a
  ];
```

The constructor `One` of arity 0 is for the 1 element of the corresponding action algebra. The following constructor `Symb` of arity 1 is the node for a generator. The type for the generator is abstract as expressed by the type parameter `'a` in the definition. The two following constructors are `Union` and `Conc` of arity 2 and describe union and concatenation operations. The last constructor `Star` is for the iteration or Kleene's star operator.

Now that we have given the datatype for the input of our algorithm, let us present the datatype for the output (automata). We choose to implement states of the automaton with integers:

```
type state = int;
```

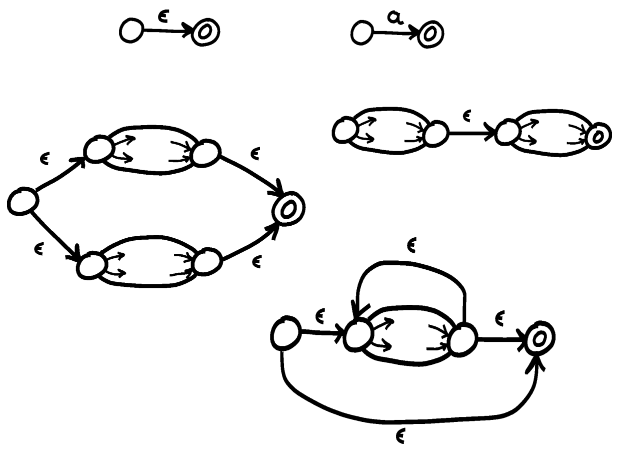
Automata obtained by Thompson's algorithm are non-deterministic and furthermore may contain ϵ-moves. We shall implement the control graph of such non-deterministic automata as a list of `fanout` pairs associating a list of labeled transitions to a state. This method amounts to encoding a set of edges $s \xrightarrow{a} s'$ or triples (s, a, s') as an association list.

```
type fanout 'a = (state × list (label 'a × state))
and label 'a = option 'a
and transitions 'a = list (fanout 'a)
;
type automaton 'a = (state × transitions 'a × state);
```

A label is of type `option 'a` because it may either be an ϵ-move of value `None` or a generator `a` of value `Some a`. Note that even if they are non-deterministic, the automata we consider have only one initial and one accepting state.

We shall instantiate the `transition` function of the control component of our machines by composing the `transitions` list component of the constructed `automaton` with the primitive `List.assoc`, as we shall show later in section 8.

Thompson's algorithm can be summarized succinctly in a graphical way:



The algorithm performs a recursive traversal of the expression and each case corresponds to a drawing. It is presented in the order of the datatype definition: 1, generator, union, concatenation and Kleene's star.

```
(* thompson: regexp 'a → automaton 'a *)
value thompson e =
  let rec aux e t n =
    (* e is current regexp, t accumulates the state space,
       n is the latest 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
;
```

The algorithm constructs the automaton from the regular expression with a single recursive traversal of the expression. States are created at each node encountered in the expression: each constructor creates two states except the concatenation `Conc` that does not create any state. Notice the invariant of the recursion: each regular subexpression builds an automaton (i, fan, f) with $0 < i < f$ and $dom(fan) = [k..f-1]$. States are allocated so that disjoint subexpressions construct disjoint segments $[i..f]$. This invariant of the `thompson` function implies that we have to add a last (empty) fanout for the final state.

```
(* thompson_alg: regexp 'a → automaton 'a *)
value thompson_alg e =
  let (i,t,f) = thompson e in
  (i, [(f,[]) :: t], f)
;
```

The function `thompson_alg` implements Thompson's algorithm in linear time and space because it performs a unique traversal of the expression.

7.2 Other algorithms

We have seen that Thompson's algorithm is linear, produces an automaton of size linear in the size of the regular expression, and can be implemented in an applicative manner. Let us mention also Berry and Sethi's algorithm [3] that computes a non-deterministic automaton (without ϵ-move), more precisely a *Glushkov* automaton. This construction is quadratic and we provided an implementation of it in ML [14]. In 2003, Ilie and Yu [15] introduced the Follow automata which are also non-deterministic automata. Actually, Champarnaud, Nicart and Ziadi [6] showed that the Follow automaton is a quotient of the one produced by the Berry-Sethi algorithm (i.e., some states are merged together) and they provide an algorithm implementing the Follow construction in quadratic time. The applicative implementation of the Berry-Sethi algorithm may be extended to yield the Follow automaton [21]. Finally, in 1996 Antimirov proposed an algorithm [2] that compiles even smaller automata than the ones obtained by the Follow construction, provided the input regular expression is presented in *star normal form* (as defined by Bruggemann [4]). The algorithm presented originally was polynomial in $O(n^5)$ but Champarnaud and Ziadi [7, 8] proposed yet another implementation in quadratic time.

It is possible to validate these various compiling algorithms using some of the algebraic laws of action algebras we presented in Section 3. In particular, using idempotency to collapse states will indicate that the corresponding construction does not preserve the notion of multiplicity of solutions. Furthermore, such a notion of multiplicity, as well as weighted automata modeling statistical properties, generalise to the treatment of valuation semi-rings, for which Allauzen and Mohri [1] propose extensions of the various algorithms. Recently Fischer et al. [10] presented a functional program implementing efficiently the matching problem for weighted regular expressions.

8 A worked-out example

We briefly discussed above how to implement as a machine a finite automaton recognizing a regular language. We may use for instance Thompson's algorithm to compile the automaton from a regular expression defining the language. This example will show that recognizing the language and generating the language are two instances of machines which share the same control component, and vary only on the data domain and its associated semantics. Furthermore, we show in the recognition part that we may compute the multiplicities of the analysed string. However, note that this is possible only because Thompson's construction preserves this notion of multiplicity.

Let us work out completely this method with the regular language defined by the regular expression $(a^*b + aa(b^*))^*$.

```
(* An example: recognition and generation of a regular language L *)

(* L = (a*b + aa(b)* )* *)
value exp =
  let a = Symb 'a' in
  let b = Symb 'b' in
  let astarb = Conc (Star a) b in
  let aabstar = Conc a (Conc a (Star b)) in
  Star (Union astarb aabstar)
;
value (i,fan,t) = thompson_alg exp
;
value graph n = List.assoc n fan
;
value delay_eos = fun () → Void
```

```
;
value unit_stream x = Stream x delay_eos
;

module AutoRecog = struct
 type data = list char;
 type state = int;
 type generator = option char;
 value transition = graph;
 value initial = [ i ];
 value accept s = (s = t);
 value semantics c tape = match c with
   [ None → unit_stream tape
   | Some c → match tape with
     [ [] → Void
     | [ c' :: rest ] → if c = c' then unit_stream rest else Void
     ]
   ];
end (* AutoRecog *)
;
module LanguageDeriv = Engine AutoRecog
;
(* The Recog module controls the output of the sub-machine
   LanguageDeriv, insuring that its input is exhausted *)
module Recog = struct
 type data = list char;
 type state = [ S1 |S2 |S3 ];
 type generator = int;
 value transition = fun
   [ S1 → [ (1,S2) ]
   | S2 → [ (2,S3) ]
   | S3 → []
   ];
 value initial = [ S1 ];
 value accept s = (s = S3);
 value semantics g tape = match g with
   [ 1 → LanguageDeriv.Fair_Engine.simulation tape
   | 2 → if tape = [] then unit_stream tape else Void
   | _ → assert False
   ];
end (* Recog *)
;
module WordRecog = Engine Recog
;
module AutoGen = struct
```

```
 type data = list char;
 type state = int;
 type generator = option char;
 value transition = graph;
 value initial = [ i ];
 value accept s = (s = t);
 value semantics c tape =
   match c with
   [ None → unit_stream tape
   | Some c → unit_stream [ c :: tape ]
   ];
end (* AutoGen *)
;
module AutoGenBound = struct
 type data = (list char × int); (* string with credit bound *)
 type state = int;
 type generator = option char;
 value transition = graph;
 value initial = [ i ];
 value accept s = (s = t);
 value semantics c (tape, n) =
   if n < 0 then Void
   else match c with
     [ None → unit_stream (tape, n)
     | Some c → unit_stream ([ c :: tape ], n-1)
     ];
end (* AutoGenBound *)
;
module WordGen = Engine AutoGen;
module WordGenBound = Engine AutoGenBound;

(* Service functions on character streams for testing *)

(* print char list *)
value print_cl l =
 let rec aux l = match l with
   [ [] → ()
   | [ c :: rest ] → let () = print_char c in aux rest
   ] in
 do { aux l; print_string "\n" }
;
value iter_stream f str =
 let rec aux str = match str with
   [ Void → ()
   | Stream v del → let () = f v in aux (del ())
   ] in
 aux str
;
value print_cl2 (tape,_) = print_cl tape
;
value cut str n =
 let rec aux i str =
   if i ≥ n then Void
   else match str with
     [ Void → Void
     | Stream v del → Stream v (fun () → aux (i+1) (del ()))
     ] in
 aux 0 str
;
value count s =
  let rec aux s n =
    match s with
    [ Void → n
    | Stream _ del → aux (del ()) (n+1)
    ] in
 aux s 0
;

print_string "Recognition␣of␣word␣‘aaaa’␣with␣multiplicity:␣";
print_int (count (WordRecog.FEM.simulation ['a' ; 'a' ; 'a' ; 'a' ]));
print_newline ();

print_string "Recognition␣of␣word␣‘aab’␣with␣multiplicity:␣";
print_int (count (WordRecog.FEM.simulation ['a' ; 'a' ; 'b' ]));
print_newline ();

(* Remark that we generate mirror images of words in L *)
print_string "First␣10␣words␣in␣~L␣in␣a␣complete␣enumeration:\n";
iter_stream print_cl (cut (WordGen.Fair_Engine.simulation []) 10);

print_string "All␣words␣in␣L␣of␣length␣bounded␣by␣3:\n";
iter_stream print_cl2 (WordGenBound.FEM.simulation ([],3));
```

We now show the output of executing the above code:

```
Recognition of word ‘aaaa’ with multiplicity: 1
Recognition of word ‘aab’ with multiplicity: 3
First 10 words in L in a complete enumeration:

b
ba
aa
```

```
baa
baa
baaa
bbaa
bb
baaaa
All words in L of length bounded by 3:
baa
bba
ba
bab
bbb
bb
aab
b
baa
baa
aa
```

The running-time of the reactive engine on these small examples is negligible. However the reactive engine performs a backtracking search that has an exponential complexity (this exponential behavior is observable using longer words in the recognition problem). Considering the generality of the relational machine model we propose, the backtracking search is an adequate technique for solving general problem. For specific problems, there might exist specific algorithms reducing the complexity; for instance, the recognition problem for automata on words can be solved in $O(mn)$ with m the size of the regular expression and n the length of the word (See Fischer et al. [10]).

Conclusion

We have presented a general model of non-deterministic computation based on a computable version of Eilenberg machines. Such relational machines complement a non-deterministic finite-state automaton over an alphabet of relation generators with a semantics function interpreting each relation functionally as a map from data elements to streams of data elements. The relations thus computed form an action algebra in the sense of Pratt. We have surveyed several algorithms that permit to compile the control component of our machines from regular expressions. The data component is implemented as an ML module consistent with an `EMK` interface. We have shown how to simulate our non-deterministic machines with a reactive engine, parameterized by a strategy. Under appropriate fairness assumptions of the strategy, the simulation is complete. An important special case is that of finite machines, for which the bottom-up strategy is complete, while being efficiently implemented as a flowchart algorithm.

We believe this applicative model of relational computing is a sound general basis for non-deterministic search processes. It encompasses the usual applications to parsing/recognition but also to generation of formal languages. It also applies to logic programming, constraints processing, database querying, and proof search for automated proof assistants. It provides a clean framework in which to develop applications to natural-language processing and similar 'artificial intelligence' problems. The flexible nature of the search strategy parameter allows one to account for statistical-optimisation techniques such as hidden Markov chains. Extensions of the action algebras to numerical operators (max, plus) should allow the adaptation of these techniques to important operations research applications such as optimisation. Finally, the ubiquitous nature of relations ought to allow the extension of this model to various models of distributed processing.

References

1. C. Allauzen and M. Mohri. A unified construction of the Glushkov, Follow, and Antimirov automata. *Springer-Verlag LNCS*, 4162:110–121, 2006.
2. V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
3. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.
4. A. Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.
5. J. A. Brzozowski. Derivatives of regular expressions. *J. Assoc. Comp. Mach.*, 11(4):481–494, October 1964.
6. J.-M. Champarnaud, F. Nicart, and D. Ziadi. From the ZPC structure of a regular expression to its follow automaton. *International Journal of Algebra and Computation (IJAC)*, 16(1):17–34, 2006.
7. J.-M. Champarnaud and D. Ziadi. From c-continuations to new quadratic algorithms for automaton synthesis. *International Journal of Algebra and Computation (IJAC)*, 11(6):707–736, 2001.
8. J.-M. Champarnaud and D. Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theoretical Computer Science*, 289(1):137 – 163, 2002.
9. S. Eilenberg. *Automata, Languages, and Machines, volume A.* Academic Press, 1974.
10. S. Fischer, F. Huch, and T. Wilke. A play on regular expressions: functional pearl. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 357–368, New York, NY, USA, 2010. ACM.
11. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27,4:797–821, 1980.
12. G. Huet. The Zen computational linguistics toolkit: Lexicon structures and morphology computations using a modular functional programming language. In *Tutorial, Language Engineering Conference LEC'2002*, 2002.
13. G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005.

14. G. Huet and B. Razet. The reactive engine for modular transducers. In K. Futatsugi, J.-P. Jouannaud, and J. 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.
15. L. Ilie and S. Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.
16. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
17. D. Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994.
18. P. Landin. The next 700 programming languages. *CACM*, 9,3:157–166, 1966.
19. V. Pratt. Action logic and pure induction. In *Workshop on Logics in Artificial Intelligence*. Springer-Verlag LNCS vol. 478, 1991.
20. B. Razet. Finite Eilenberg machines. In O. Ibarra and B. Ravikumar, editors, *Proceedings of CIIA 2008*, pages 242–251. Springer-Verlag LNCS vol. 5148, 2008. `http://gallium.inria.fr/~razet/fem.pdf`
21. B. Razet. *Machines d'Eilenberg Effectives.* PhD thesis, Université Denis Diderot (Paris 7), 2009.
22. B. Razet. Simulating finite Eilenberg machines with a reactive engine. *Electronic Notes in Theoretical Computer Science*, 229(5):119 – 134, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).
23. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.