The Reactive Engine for Modular Transducers

Gérard Huet and Benoît Razet

INRIA Rocquencourt,
BP 105, 78153 Le Chesnay Cedex, France

Abstract. This paper explains the design of the second release of the Zen toolkit [5–7]. It presents a notion of reactive engine which simulates finite-state machines represented as shared *aums* [8]. We show that it yields a modular interpreter for finite state machines described as *local* transducers. For instance, in the manner of Berry and Sethi, we define a compiler of regular expressions into a scheduler for the reactive engine, chaining through aums labeled with phases — associated with the letters of the regular expression. This gives a modular composition scheme for general finite-state machines.
Many variations of this basic idea may be put to use according to circonstances. The simplest one is when aums are reduced to dictionaries, i.e. to (minimalized) acyclic deterministic automata recognizing finite languages. Then one may proceed to adding supplementary structure to the aum algebra, namely non-determinism, loops, and transduction. Such additional choice points require fitting some additional control to the reactive engine. Further parameters are required for some functionalities. For instance, the local word access stack is handy as an argument to the output routine in the case of transducers. Internal virtual addresses demand the full local state access stack for their interpretation.
A characteristic example is provided, it gives a complete analyser for compound substantives. It is an abstraction from a modular version of the Sanskrit segmenter presented in [9]. This improved segmenter uses a regular relation condition relating the phases of morphology generation, and enforcing the correct geometry of morphemes. Thus we obtain compound nouns from iic*.(noun+iic.ifc), where iic and ifc are the respectively prefix and suffix substantival forms for compound formation.

Dedicated to Joseph Goguen for his 65th birthday

1 Regular morphology

We first consider the simplest framework for finite automata, where the state transition graph is a dictionary structure (lexical tree or trie). Such structures represent acyclic deterministic finite-state automata, with maximal sharing of initial paths. Every state is accessible from the initial state, and we may also assume that every state is on an accepting path. When we minimize the tree as a dag, we obtain the corresponding minimal deterministic automaton. Such

automata recognize finite languages. They are adequate for representing the lexicons of natural languages.

In a framework of generative morphology, we want to model the construction of lexemes from smaller chunks called morphemes: radical stems, prefixes and suffixes. It is convenient to sort the morphemes into categories, and to enforce structural conditions on these categories, restricting the geometry of lexemes. For instance, we may describe this geometry by a regular expression over the alphabet of lexical categories. The language generated by the regular expression, substituting each category by its corresponding morpheme lexicon, is recognized by a modular reactive engine, which chains the morphemes dictionary lookup with transitions corresponding to the regular expression recognizer. We shall use for this setup variants of the compiling algorithm of Berry and Sethi [2].

1.1 Automaton interface

We use as algorithmic description language Pidgin ML, a core applicative subset of Objective Caml. Thus our algorithms may be read as rigorous higher-order inductive definitions, while being directly executable, in the spirit of *literate programming*.

We first recall the basic structures of the Zen toolkit [5].

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

```
module Word : sig

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

Thus the basic morphology operations will rely on list processing, and *not* on string processing (and certainly not on encoding formats such as Unicode-UTF8, which are meant for data exchange portability and should not be used for core computation).

Here is the interface to our simplistic automata, reduced to deterministic transitions over a lexicon tree. Each state is labeled with a boolean (indicating whether or not it is an accepting state), and points to the list of its successor states, labeled with a letter.

```
module Auto : sig

type auto = [ State of (bool × deter) ]
and deter = list (Word.letter × auto);
end;
```

We assume that at most one transition issued from a given state is labeled with a given letter. The datatype *auto* is here isomorphic to lexical trees, or dictionary, also called *tries*. We may also assume that dead alleys, i.e. states which do not have an accepting node as a substructure, are ruled out. Thus the

contraction of the tree as a dag, using for instance the corresponding instance of the *sharing functor* [5], yields the minimal automaton that recognizes the finite language stored in the dictionary.

1.2 Dispatching

We call *phases* the lexical categories, which constitute the alphabet of the regular expression defining the morphological geometry. We compile this regular expression using the Berry-Sethi method, which linearizes the expression, and computes the local automaton associated to this linearization [2, 3].

We recall that local automata (also called Glushkov automata) are finite automata such that all transitions labeled with a given letter lead to the same state, characteristic of this letter. States may thus be named with letters, here phases. It is this locality condition which is a key to modularity.

A local automaton is described by an *initial* phase, a set of *terminal* phases, here represented as a boolean function over phases, and a *dispatch* transition function, mapping each phase to a set of following phases, sequentialized here as a list. In the notations of [2], *initial* is called *1*, *dispatch* is called *follow*, and *terminal* is implicit from the use of an endmarker symbol. In the terminology of Eilenberg [4], the set of non-empty words recognized by a local automaton is a local set over phases.

In the Zen toolkit implementation, the *Dispatch* module is actually generated by meta-programming, i.e. it compiled from the regular expression, as we shall explain in section 4.

1.3 Scheduling

We are now ready to start the description of the reactive engine, as a functor taking a module *Dispatch* as parameter, and using its *dispatch* function as a local scheduler. Here is the corresponding specification of our *React* module. We assume the utility programming functions *fold_right* (list iterator), *assoc*, *length*, *mem*, etc. from the *List* standard library.

```
module React
  (Dispatch: sig
    type phase = α;
    value transducer : phase → auto;
    value initial : phase;
    value terminal : phase → bool;
    value dispatch : phase → list phase;
    end) = struct
type input = Word.word
and backtrack = [ Advance of phase and input ]
and resumption = list backtrack;
```

A *resumption* value stores as a datum what is necessary to resume our reactive engine as a coroutine.

The scheduler gets its phase transitions from *dispatch*. It respects the order of dispatching.

```
value schedule phase input cont =
  let add phase cont = [ Advance phase input :: cont ] in
  fold_right add (dispatch phase) cont;
```

1.4 React

The reactive engine originates from the Sanskrit segmenter described in [9], generalized to the framework of mixed automata defined in [8].

Here we have a much simpler framework, since we do not have transducer output, but we get a *modular* interpreter, driven by the phase scheduler.

In the following definition, *phase* is the current phase, *input* is the input tape represented as a *word*, *back* is the backtrack stack of type *resumption*, and *state* is the current state of type *auto*. We favor deterministic transitions within a phase to non-deterministic transitions to the next phase(s). Within a phase, we favor longer words over shorter ones. Phase transitions are effected in *dispatch* order. We have a mutual inductive definition between the reactive engine, reading forward, and the continuation manager, backtracking on failure.

```
exception Finished;

value rec react phase input back state = match state with
  [ State (b,det) →
  let deter cont = match input with
        [ [] → continue cont
        | [ letter :: rest ] →
          try let state' = assoc letter det in
              react phase rest cont state'
          with [ Not_found → continue cont ]
        ] in
  if b (* accepting *) then
       if input=[] (* end of input *) then
          if terminal phase then back (* solution found *)
          else continue back
       else let cont = schedule phase input back in
            deter cont
  else deter back
  ]
and continue = fun
  [ [] → raise Finished
  | [ resume :: back ] → match resume with
      [ Advance phase input →
           react phase input back (transducer phase)
      ]
  ];
```

1.5 Usage

The initialization of the reactive engine consists in setting the backtrack stack to the single initial state given by *Dispatch.initial*, *input* being initialized to the full sentence:

```
value init_react sentence = [ Advance initial sentence ];
```

We may now recognize a string as belonging to the rational language described by the regular expression by calling the reactive continuation manager with this initial resumption:

```
value react1 sentence = continue (init_react sentence);
```

If the sentence belongs to the language, *react1* will return with a resumption value, otherwise it will throw the exception *Finished*. The resumption value is not of use in this simple model, where the interpreter is used as a mere recognizer. In more elaborate versions below, *react* may be used as a coroutine in order to compute a stream of transductions.

Note that classical formal languages theory abstracts a language as a set of words, or occasionally as a multiset (hiding structural idempotence) when multiplicities matter. Here we hide structural commutativity as well, obtaining streams of solutions, where computational details such as fairness, essential for completeness, may be revealed and discussed.

1.6 Correctness, completeness

Let us be given a module *Dispatch* by its components *phase* (an ordered list of discrete phase values defining the alphabet), *initial* (the initial phase), and functions *transducer* : phase → auto, *terminal* : phase → bool and *dispatch* : phase → list phase.

Let $L(\phi)$ be the language recognized by the automaton $transducer(\phi)$, for a given phase ϕ. We assume that $L(initial)$ is the singleton $\{\epsilon\}$ where ϵ is the empty word [], that $L(\phi)$ does not contain ϵ for any other phase ϕ, and that for every phase ϕ the list $dispatch(\phi)$ does not contain $initial$. These invariants will be enforced by the Berry-Sethi compiler presented in section 4.

Let us say that a sequence $((\phi_1, w_1), ... (\phi_n, w_n))$ is a valid *analysis* of a given word w whenever, taking $\phi_0 = initial$, we get $w = w_1 \cdot w_2 \cdot ... \ w_n$ with $(0 < i \leq n)\ w_i \in L(\phi_i)$, $(0 \leq i < n)\ \phi_{i+1} \in dispatch(\phi_i)$, and $terminal(\phi_n) = True$. For $i > 0$, we know from the assumptions above that $L(\phi_i)$ does not contain the empty word, so there is a finite number of such analyses.

We define a total ordering on analyses by the lexicographical ordering generated by $(\phi, w) < (\phi', w')$ iff either ϕ precedes ϕ' in the common dispatch list where ϕ and ϕ' belong, or else $\phi = \phi'$ and w' is a strict initial prefix of w.

The correction and completeness of the *react* algorithm may be established by proving that it generates the set of valid analyses of an input word w in the sense that it implicitly builds a sequence of pairs of analyses of w and resumptions $((\alpha_1, r_1), ...(\alpha_N, r_N))$ such that, taking $r_0 = init\_react\ w$, for each i $(0 \leq i < N)$

the evaluation of *continue* r_i terminates with value r_{i+1}, and the evaluation of *continue* r_N raises the exception *Finished*. Furthermore the list $(\alpha_1, ...\alpha_N)$ contains all the valid analyses of w, listed increasingly with respect to the above ordering.

We shall not give a formal proof of this rather fastidious property, which can be established by computational induction.

We remark that this argument makes explicit the fact that, within a given phase, we search for longer partial solutions before shorter ones. This is a rather arbitrary heuristic, which is convenient for the segmenting application.

2 Modular aums

So far our automata have been mere recognizers for finite sets of words, i.e. dictionaries. Chaining them through phases, we may for instance model simple segmentation problems, where a sentence is defined as a list of words separated by blanks or punctuation signs, and words are defined as compounds of morphemes, according to prefix, suffix, or other finite-state regimes. Such a segmenter may be composed with a tagger, when the word dictionaries are decorated with morphological derivation annotations, using the structure of *revmaps* [5], which allows efficient sharing of morphological regularities.

We now allow more complex automata for the various phases. For instance, we may allow a notion of transition with virtual addresses, allowing both non-deterministic moves (including ϵ-transitions), and cycles.

Virtual addresses, as opposed to pointers and explicit cyclic structures, provide a declarative mechanism respecting sharing. In the original presentation of aums [8], two varieties of virtual addresses are proposed: absolute addresses, indexing a state by its absolute access path in the forest of deterministic skeletons, and relative addresses, indexing a state in the current covering trie by the shorthest path in the tree, encoded as a *differential word* pairing a natural number (how many levels in the tree you should go up) with a word (indexing the target state down from the closest common ancestor). These differential words are used for instance in the *revmap structure*, to store the reverse morphology.

In the next section, we shall ignore relative addresses, which necessitate a slightly more complex apparatus for their proper evaluation, since sharing makes ambiguous the inheritance relation, and thus access paths must be maintained in the automaton structure for proper interpretation. We shall present first simple absolute addresses. Furthermore, the role of the forest index will be played by the phase: to each phase corresponds a unique *auto* structure, covering all the states pertaining to this phase.

2.1 Mixed automata with virtual absolute addresses

A transition (w, v) recognizes word w on the input tape (the "guard" of the transition), and jumps to the state absolutely adressed by v in the next phase.

```
module Auto : sig

type transition = (Word.word × Word.word)
and choices = list transition;

type auto = [ State of (deter × choices) ]
and deter = list (Word.letter × auto);
end;
```

We take as convention that the state $State(d, c)$ is accepting iff c is not empty. We now define acceptance as the condition on external transitions (w, v) when the input is empty, the (next) phase is terminal, and the access parameter v verifies a *final* condition which we shall not precise further. Typically, v is final if it is empty or if it consists in a special end of sentence marker.

2.2 Service routine

Our resumptions are now more complex, since we have non-deterministic choice points:

```
type backtrack =
  [ Choose of phase and input and auto and choices
  | Advance of phase and input and word
  ]
and resumption = list backtrack;

exception Finished;
```

Here are two service routines to manage guard management.

```
exception Guard;
value rec advance n w = if n = 0 then w else match w with
  [ [] → raise Guard
  | [ _ :: tl ] → advance (n−1) tl
  ];
```

Thus *advance* n $[a_1;\ ...\ a_N] = [a_p;\ ...\ a_N]$, where $p = N - n$, whenever $n \leq N$; otherwise the exception *Guard* is raised.

```
(* [access : phase → word → auto ] *)
value access phase = acc (transducer phase)
   where rec acc state = fun
      [ [] → state
      | [ c :: rest ] → match state with
        [ State (deter, _) →
          acc (List.assoc c deter) rest
        ]
      ];
```

2.3 React for aums

We use a similar *schedule* function as previously, it now stores the v access path for the next phase transition.

```
value schedule phase input v cont =
  let add phase cont = [ Advance phase input v :: cont ]
  in fold_right add (dispatch phase) cont;
```

We are now ready to present the reactive engine. It consists in three simultaneous inductions, the main one *react* managing the deterministic search, while stacking non-deterministic choice points, the second *choose* managing non-deterministic jumps, and the third *continue* backtracking in case of dead end. We favor deterministic transitions over non-deterministic ones.

```
(* phase is the parsing phase,
   input is the input tape represented as a word,
   back is the backtrack stack of type resumption,
   state is the current state of type auto *)
value rec react phase input back state =
  match state with
  [ State (det,choices) →
    (* we explore the deterministic space first *)
    let cont = if choices=[] then back else
        [ Choose phase input state choices :: back ]
    in match input with
       [ [] → continue cont
       | [ letter :: rest ] →
           try let next_state = assoc letter det in
               react phase rest cont next_state
           with [Not_found → continue cont]
       ]
  ]
and choose phase input back state = fun
  [ [] → continue back
  | [ (w,v) :: others ] →
    let cont = if others=[] then back else
        [ Choose phase input state others :: back ]
    in try let tape = advance (length w) input in
           if tape = [] (* input finished *) then
              if terminal phase && final v then cont
              else continue cont
           else continue (schedule phase tape v cont)
       with [Guard → continue cont]
  ]
and continue = fun
  [ [] → raise Finished
  | [ resume :: back ] → match resume with
```

```
      [ Choose phase input state choices →
          choose phase input back state choices
      | Advance phase input word →
          try let next_state = access phase word
              in react phase input back next_state
          with [Not_found → continue back]
      ]
  ];
```

Finally, here is the initialisation routine, building the initial resumption:

```
value init_react input = [ Advance initial input [] ];
```

As previously, we may recognize a sentence using:

```
value react1 sentence = continue (init_react sentence);
```

2.4 Correctness, completeness

Similarly to the previous section, we may prove the correctness and completeness of the construction, provided the guard w of each non-deterministic transition is non-empty. We may refine this condition as follows.

Definition: Guard condition. There is no cycle of transitions of an aum all of which have an empty guard: $(\epsilon, w_1); (\epsilon, w_2); ...(\epsilon, w_n)$. By cycle we mean that, for some access word w_0 in the current phase ϕ_0 leading in *transducer(ϕ_0)* to state σ_0, σ_0 has among its choices (ϵ, w_1), ϕ_1 in *dispatch(ϕ_0)* with w_1 leading in *transducer(ϕ_1)* to state σ_1, etc, until $\sigma_n = \sigma_0$.

We claim that *react* terminates on an input word whenever the guard condition is verified. Note that this is a global condition on the family of aums, which requires the knowledge of the phase transition relation, but which may be checked in time linear in the cumulated size of the aum family.

3 Modular aum transducers

We now give the final refinement of our construction, with aums having both local and global virtual addresses.

3.1 Transducers

```
module Auto : sig

type continuation = (Word.word × Word.word)
and transition =
  [ External of (Word.word × continuation)
  | Internal of (Word.word × Word.delta)
  ];
```

```
type auto = [ State of (deter × choices) ]
and deter = list (Word.letter × auto)
and choices = list transition;
end;
```

An internal transition $Internal(w, d)$ recognizes w on the input tape and jumps to the state relatively addressed by d within the same phase. This uses the notion of *differential word* [5] from module Word:

```
type delta = (int × word); (* differential words *)
```

A differential word is a notation permitting to retrieve a word w from another word w' sharing a common prefix. It denotes the minimal path connecting the words in a trie, as a sequence of ups and downs: if $\delta = (n, u)$ we go up n times and then down along word u. In order to interpret the n part, we need to keep the stack of states leading locally to the current state. We keep along this *stack* the corresponding word path as well — this is useful as a parameter to the output computation.

An external transition $External(w, c)$ recognizes w on the input tape and executes the continuation c in a following phase. A continuation (u, v) returns words u as output parameter and v as access parameter in the next phase transducer.

As above we define acceptance as the condition on external transition when the input is empty, the phase is terminal, and the access parameter v verifies a *final* condition which we shall not precise further.

3.2 Modular transducers

We now produce output, as words labeled by their phase.

```
type input = Word.word
and output = list (phase × Word.word);
```

The access stack has a letter component and a state component. The state component is necessary to interpret the part of the internal virtual address which concerns going up, whereas the letter component, i.e. the absolute name of the state in the current phase is useful for computing the transducer output.

```
type stack = list (Word.letter × auto);

type backtrack =
  [ Choose of phase and input and output
          and auto and stack and choices
  | Advance of phase and input and output and Word.word
  ]
and resumption = list backtrack;
```

Since the *Advance* resumption has now an output component and an access component (anticipating a prefix of the next phase component), we parameterize the scheduler accordingly:

```
value schedule phase input output access cont =
  let add phase cont =
    [ Advance phase input output access :: cont ]
  in fold_right add (dispatch phase) cont;
```

The service routine *access* manages the access stack, the functions *pop* and *push* are used to interpret internal jumps.

```
(* access : phase → word → ( auto × stack ) *)
value access phase = acc (transducer phase) []
  where rec acc state stack = fun
  [ [] → (state,stack)
  | [ c :: rest ] → match state with
    [ State (deter,_) →
      acc (assoc c deter) [ (c,state) :: stack ] rest
    ]
  ];

value rec pop n state stack =
  if n=0 then (state,stack)
  else match stack with
     [ [] → raise (Failure "Wrong Internal jump")
     | [ (_,st) :: rest ] → pop (n-1) st rest
     ]
and push w state stack = match w with
     [ [] → (state,stack)
     | [ c :: rest ] → match state with
       [ State (deter,_) →
         push rest (assoc c deter) [ (c,state) :: stack ]
       ]
     ];

value jump (n,w) state stack =
  let (state0,stack0) = pop n state stack
  in push w state0 stack0;
```

We provide the access stack as an output parameter via an extracting routine:

```
value extract stack (_,(u,_)) =
  fold_left unstack u stack
    where unstack acc (c,_) = [ c :: acc ];
```

3.3 Modular reacting transducers

We have a similar structure of three mutually recursive functions, but now *choose* has two cases, for the two transition constructors.

```
value rec react phase input output back stack state =
  match state with
  [ State (det,choices) →
    let cont = if choices=[] then back else
     [Choose phase input output state stack choices :: back]
    in match input with
      [ [] → continue cont
      | [ letter :: rest ] →
        try let state' = assoc letter det
            and stack' = [ (letter,state) :: stack ] in
            react phase rest output cont stack' state'
        with [ Not_found → continue cont ]
      ]
  ]
and choose phase input output back state stack = fun
  [ [] → continue back
  | [ External((w,(u,v)) as rule) :: others ] →
    let cont = if others=[] then back else
     [Choose phase input output state stack others :: back]
    in try let tape = advance (length w) input
           and out = [(phase,extract stack rule) :: output]
           in if tape = [] (* input finished *) then
                if terminal phase && final v then (out,cont)
                else continue cont
              else continue (schedule phase tape out v cont)
       with [ Guard → continue cont ]
  | [ Internal(w,delta) :: others ] →
    let cont = if others=[] then back else
     [Choose phase input output state stack others :: back]
    in try let tape = advance (length w) input
           and (state',stack') = jump delta state stack
           in react phase tape output cont stack' state'
       with [ Guard → continue cont ]
  ]
and continue = fun
  [ [] → raise Finished
  | [ resume :: back ] → match resume with
      [ Choose phase input output state stack choices →
         choose phase input output back state stack choices
      | Advance phase input output word →
         try let (state',stack') = access phase word
             in react phase input output back stack' state'
         with [ Not_found → continue back ]
      ]
  ];
```

3.4 Correctness, completeness

The definitions of trace and analysis may be extended to the case of transducers, and the correctness and completeness of our engine may be formally proved in the sense that all transductions of the input word are properly generated, for a notion of left-to-right transduction. We omit here the full formal development.

In the case of non overlapping junction transductions, as defined in [9], the construction simplifies, since *Internal* transitions are not needed. The proofs of termination, correctness and completeness of the reactive engine are carried out in full in [9], for the simple case of one phase junction relations verifying a non-overlapping criterion. This criterion allows parallel computation of the relation along phases, without the need to cascade the transductions. Furthermore, such relations are invertible, and the reactive engine may thus be used to invert euphony and return segmentation solutions, even when the euphony relation is not length-preserving.

Other variations may be considered, since the presence or absence of output transitions is orthogonal to the structure of virtual addresses. We have considered virtual addresses of two kinds, internal and external. We may also imagine other encodings of jumps, potentially relevant for specific applications. For instance, specific encodings, relying on the fact that the underlying alphabet is boolean, may be used to represent boolean circuits, in the manner of BDD structures.

The general problem of compiling an arbitrary finite-state machine description into some variety of our *aum* structures is not addressed in the current paper. This problem has many degrees of freedom, since there is a choice between mapping state transitions into the deterministic skeleton, on one hand, and the non-deterministic choices sequences, on the other; in the latter case, there is a further choice between External and Internal jumps. Finally, the partition into phases may be more or less coarse, and extra encoding letters, disjoint from the input alphabet, may be used to attach orphan states. We should not expect one uniform best solution to this problem anyway, and compiling strategies may well depend on the application domain.

Remark.
In [9], section 8.1, the recursive call from *choose* calls *react* with *occ* parameter v, instead of $rev\ v$ as effected above for *next_stack*. This is a local optimisation for the case of *sandhi*, where the junction rules are such that the length of component v is at most 1.

4 Dispatch synthesis from regular expressions

We now explain how to synthesize the *dispatch* function from a regular expression representation of the phase language, using the Berry-Sethi algorithm [2]. The basic idea is that we compose a number of finite automata/transducers, each named with a *phase*. Phases are the letters of an alphabet, and we define the admissible joint behaviour of our automata as a rational language over the phase alphabet, specified by a regular expression.

4.1 Regular expressions and their linearization

Here is the type of regular expressions. The type parameter α is used to abstract from the symbol representation.

```
type regexp α =
  [ One
  | Symb of α
  | Union of regexp α and regexp α
  | Conc of regexp α and regexp α
  | Star of regexp α
  | Epsilon of regexp α
  | Plus of regexp α
  ];
```

We use a specific constructor *Plus* rather than defining R^+ as the macro $R \cdot R^*$, because of the blow-up due to its non-linearity.

We mark symbols with an integer to linearize the regular expression.

```
type marked α = (α × int);
```

A symbol s is mapped to $(s, 0)$ if it occurs only one, and to $(s, 1)$, $(s, 2)$, etc. otherwise. Marked symbols are used as states of the recognizing automaton. The type *local* represents local automata, in the sense of Eilenberg, as a 4-tuple defining its initial state, the other states, the transitions, and the terminal states:

```
type local α =
  ( marked α × list (marked α)
  × list (marked α × list (marked α))
  × list (marked α)
  );
```

We skip the details of the linearization function *mark*, which is straightforward. The function *mark* takes as argument a *regexp* α, and returns a pair of type *regexp(marked* α*)* $\times$ *list(marked* α*)*, consisting of the marked expression, and the list of marked symbols which will be used as states of the local automaton.

4.2 The Berry-Sethi compiler

We basically follow the construction given in [2], with the addition of the *Plus* operation. We need an intermediate structure of *discriminating* regular expressions, which makes explicit whether the associated rational language contains the empty word ϵ or not.

```
type d_regexp α =
  [ DOne
  | DSymb of α
  | DUnion of bool and d_regexp α and d_regexp α
  | DConc of bool and d_regexp α and d_regexp α
  | DStar of d_regexp α
```

```
  | DEpsilon of d_regexp α
  | DPlus of bool and d_regexp α
  ];
```

We can tell in unit time this property with function *delta*, and translate in linear time a regexp in a discriminating regexp with function *discr*.

```
value delta = fun
  [ DOne → True
  | DSymb _ → False
  | DUnion b _ _ | DConc b _ _ → b
  | DStar _ | DEpsilon _ → True
  | DPlus b _ → b
  ];

(* discr : regexp α → d_regexp α *)
value rec discr = fun
  [ One → DOne
  | Symb s → DSymb s
  | Union e1 e2 →
      let de1 = discr e1 and de2 = discr e2 in
      DUnion (delta de1 || delta de2) de1 de2
  | Conc e1 e2 →
      let de1 = discr e1 and de2 = discr e2 in
      DConc (delta de1 && delta de2) de1 de2
  | Star e → DStar (discr e)
  | Epsilon e → DEpsilon (discr e)
  | Plus e →
      let de = discr e in
      DPlus (delta de) de
  ];
```

The core of the algorithm is the computation of sets *first*, *follow* and *last*.

```
(* first : list α→ d_regexp α→ list α *)
value rec first l = fun
  [ DOne → l
  | DSymb d → [ d :: l ]
  | DUnion _ e1 e2 → first (first l e2) e1
  | DConc _ e1 e2 →
      if delta e1 then first (first l e2) e1
      else first l e1
  | DStar e | DEpsilon e | DPlus _ e → first l e
  ];

(* follow : α → regexp α → list (α × list α) *)
value follow initial exp =
  let rec f1 exp l fol =
```

```
    match exp with
    [ DOne → fol
    | DSymb d → [ (d,l) :: fol ]
    | DUnion _ e1 e2 →
       let fol2 = f1 e2 l fol in f1 e1 l fol2
    | DConc _ e1 e2 →
       let fol2 = f1 e2 l fol in
       let l1 = if delta e2 then first l e2
                else first [] e2 in
       f1 e1 l1 fol2
    | DStar e | DPlus _ e →
       let l_res = first l e in
       f2 e l_res fol
    | DEpsilon e → f1 e l fol
    ]
  and f2 exp l fol = (* (first [] exp) already in l *)
    match exp with
    [ DOne → fol
    | DSymb d → [ (d,l) :: fol ]
    | DUnion _ e1 e2 →
       let fol2 = f2 e2 l fol in f2 e1 l fol2
    | DConc _ e1 e2 →
       let b1 = delta e1
       and b2 = delta e2 in
       if b1 (* l1 and l2 in l *)
       then if b2
            then f2 e1 l (f2 e2 l fol)
            else f1 e1 (first [] e2) (f2 e2 l fol)
       else if b2
            then f2 e1 (first l e2) (f1 e2 l fol)
            else f1 e1 (first [] e2) (f1 e2 l fol)
    | DStar e | DEpsilon e | DPlus _ e → f2 e l fol
    ] in
  let fol_sets = f1 exp [] []
  and initials = first [] exp in
  [ (initial,initials) :: fol_sets ];
```

Functions *f1* and *f2* both compute the follow sets of Berry-Sethi but with different assertions on their arguments; precisely, a call *(f1 exp l fol)* is such that first elements of *exp* are not in *l*, and the contrary assertion obtains for *f2*. Thus we never attempt to add elements already present in *l*, which maintains a constant cost of adding an element in *l*.

```
(* last : α → regexp α → list α *)
value last initial e =
  let rec last_rec l = fun
    [ DOne → l
```

```
    | DSymb d → [ d :: l ]
    | DUnion _ e1 e2 →
        last_rec (last_rec l e2) e1
    | DConc _ e1 e2 →
        if delta e2 then last_rec (last_rec l e2) e1
        else last_rec l e2
    | DStar e | DEpsilon e | DPlus _ e → last_rec l e
    ] in
  let l = last_rec [] e in
  if delta e then [ initial :: l ] else l;
```

Now we have all the ingredients to compile a regular expression:

```
(* compile : marked α → regexp α → local α *)
value compile initial exp =
  let (exp_m, states) = mark exp in
  let exp_d = discr exp_m in
  let fol = follow initial exp_d
  and lasts = last initial exp_d in
  (initial, states, fol, lasts);
```

4.3 Parametric regular expressions

We now define systems of regular expressions over parametric alphabets whose symbols are associated to aums. Meta-variables allow sharing in such descriptions. We skip the details of the syntax, and present just an example of such a finite machine description, actually a subproblem of Sanskrit morphology, namely noun phrases representing compound substantives.

```
initial init epsilon_aum

alphabet noun ; iic ; ifc end

automaton Disp
  node SUBST = iic* . (noun | iic.ifc)
end
```

Here we specify that the initial phase is called `init`, that the user must provide a value `epsilon_aum` for the aum recognizing just the empty word, as well as aum values `noun`, `iic` and `ifc` for recognizing the corresponding languages. We are interested in the language `iic* . (noun | iic.ifc)`. In the intended application, *SUBST* is the language of substantive forms, containing *noun* forms as well as compounds, formed with prefix *iic* forms which may be iterated, and suffix *ifc* forms.

We skip the details of the parsing of such a description. In the current syntax, we allow systems of regular expressions, allowing sharing, and the compiler unfolds the system into a flattened expression.

We use the meta-programming facilities provided by the Camlp4 preprocessor, which allows macro-generation of an Ocaml program at the level of abstract syntax. Skipping the details of this meta-programming, we obtain mechanically, for the above example, the following module text.

```
module Automata (Auto : sig type auto = 'a; end) =
  struct
    type auto_vect =
      { epsilon_aum : Auto.auto;
        noun : Auto.auto; iic : Auto.auto; ifc : Auto.auto };
    module Disp (Fsm : sig value autos : auto_vect; end) =
      struct
        type phase =
          [ Init | Iic1 | Noun | Iic2 | Ifc ];
        value transducer = fun
          [ Init → Fsm.autos.epsilon_aum
          | Iic1 → Fsm.autos.iic
          | Noun → Fsm.autos.noun
          | Iic2 → Fsm.autos.iic
          | Ifc → Fsm.autos.ifc
          ];
        value dispatch =
          fun
          [ Init → [ Iic1; Noun; Iic2 ]
          | Iic1 → [ Iic1; Noun; Iic2 ]
          | Noun → []
          | Iic2 → [ Ifc ]
          | Ifc → []
          ];
        value initial = Init;
        value terminal phase = List.mem phase [ Noun; Ifc ];
      end;
  end;
```

We now have all the components we wish to assemble, since the module instanciation `(Automata Auto)`, for `Auto` one of the aum description modules given in the previous sections, creates a module `Dispatch=(Disp Fsm)` having the right functionality, with module `Fsm` holding the aum implementations. In this simple example these implementations are the various lexicons corresponding to the respective lexical categories. In the Sanskrit platform, these aums are decorated with non-deterministic transitions (using external addressing) corresponding to sandhi prediction.

Remarks. 1. During the Berry-Sethi compiling process, the candidate regular expression is linearized when a phase occurs more than once. However, the corresponding automata are shared via the *transducer* component, recovering the proper sharing.

2. Our Sanskrit platform[1] now uses this modular methodology, which enforces the right geometry for morphological *chunks*, taking care of preverb affixes, proper recognition of compound forms and periphrastic verbal constructions, and proper analysis of absolutive forms (with suffixes in *-tvā* for roots and *-ya* for verbs admitting preverbs).
3. As usual, we may augment our automata descriptions with weights reflecting (possibly conditional) probabilities in order to get stochastic automata whose behaviour reflects hidden Markov chains in the data. Note that the correctness criteria are invariant with the permutation of choices induced by priority selection according to these weights.

4.4 A variant using Antimirov's compiling algorithm

V. Antimirov proposed in [1] another algorithm for compiling regular expressions, using a notion of *partial derivative*. This algorithm produces automata that may be significantly smaller than the ones obtained by the Berry-Sethi algorithm. Such automata do not have the locality condition, and now the modularity of the construction obtains by a more complex mapping, since the transducer invocation does not simply depend on the states, but on the transitions. We shall not develop further this variant construction in this paper.

5 Conclusion

We have presented a methodology for constructing finite-state machines, such as finite automata and transducers, in a modular way. Regular expressions over an alphabet of phases express a composition of machines under a finite-state-controlled constraint. This corresponds to considering a regular expression not as the mere denotation of a rational language over the alphabet of its symbols seen as string generators, but rather as a rational polynomial over its symbols, abstracting themselves rational sets. The algebraic property of closure of rational sets over substitution (mapping symbols to rational sets), together with the local automaton representation of finite-state machines, provide the natural foundation for the modular composition of finite-state machines.

Our mechanism allows the controlled interaction of machines compiled as mixed automata (aums). This is useful for instance for shallow parsing in computational linguistics applications. For the Sanskrit platform built by the first author, this allows to build a tagger composing machines which invert phonology (sandhi analysis) and morphology, with separate machines for distinct lexical classes, constrained by the geometrical conditions defining admissible compounds, preverb management, and periphrastic constructions with auxiliary verbs.

Our design exploits and justifies our functional programming methodology as follows:

[1] `http://sanskrit.inria.fr/`

- Applicative programming leads to robust well-structured programs, amenable to formal proofs and to journal publication, in the spirit of literate programming — all our programs are rigorously expressed as inductive definitions over higher-order types.
- Functionality is essential to the concise expression of powerful control paradigms such as continuations, essential for the definition of coroutine interpreters for non-deterministic search.
- Modularity of the programming language is the essence of the parametricity underlying algebraic closure operations, and thus is an essential abstraction paradigm.
- Powerful macro-generation mechanisms lead to an effective meta-programming methodology, tailoring general algorithms to the specific needs of applications.
- Despite this very high-level view of software architecture, the resulting programs are efficient enough for their integration in real size applications, as witnessed by their use in computational linguistic platforms [9].

References

1. V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155:291–319, 1996.
2. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
3. J. Berstel and J.-E. Pin. Local languages and the Berry-Sethi algorithm. *Theoretical Computer Science*, 155:439–446, 1996.
4. S. Eilenberg. *Automata, Languages, and Machines, volume A*. Academic Press, 1974.
5. G. Huet. The Zen computational linguistics toolkit. Technical report, ESSLLI Course Notes, 2002. `http://pauillac.inria.fr/~huet/ZEN/esslli.pdf`
6. 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.
7. G. Huet. Linear contexts and the sharing functor: Techniques for symbolic computation. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*. Kluwer, 2003. `http://pauillac.inria.fr/~huet/PUBLIC/DB.pdf`
8. G. Huet. Automata mista. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 359–372. Springer-Verlag LNCS vol. 2772, 2004. `http://pauillac.inria.fr/~huet/PUBLIC/zohar.pdf`
9. G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005. `http://pauillac.inria.fr/~huet/PUBLIC/tagger.pdf`.
10. E. Roche and Y. Schabes. *Finite-State Language Processing*. MIT Press, 1997.
11. R. Sproat. *Morphology and Computation*. MIT Press, 1992.