

A compiler of regular expressions to NFAs

Michael Day
YesLogic Pty Ltd

Vladimir Komendantsky
YesLogic Pty Ltd

12th May 2009

Abstract

Working with the proof assistant Coq, we present a proof of constructive existence of a non-deterministic finite automaton (NFA) for any given regular expression. In programming terms, the constructive proof extracted to a functional programming language yields a compiler of regular expressions to NFAs. Our goal is to build a proof library for constructive reasoning and programming with practical regular expressions (POSIX, Perl, etc.).

1 Introduction

In this paper we present a part of our formal proof effort [8] which is aimed at creating a convenient environment for programming correct-by-construction algorithms for practical regular expressions adopted by operating systems and programming languages. While there exist many such standardisations (POSIX, Perl, Emacs, Java, etc.), we adopt a unifying approach and treat mathematical invariants of regular expressions rather than side with any particular specification. The present paper concerns the very basic regular expressions with no additional operators such as shuffle or backreferences.

There seem to be several related developments available [9, 5]. Still, we found it more appropriate to initiate our own because our aim requires certain features (for example, partial derivatives of regular expressions [1]) that have not been mechanised yet in any system. Moreover, we have not been able to find a working library of non-deterministic finite automata in the present version of the proof assistant Coq [6] which is at the moment our preferred system. We employ dependent types up to a considerable extent. This especially concerns the type of regular expressions.

2 Methodology

To build the model of a recursive program we employ definitions and proofs by structural induction. In Coq, one defines structural inductive types using the command `Inductive` that has the following syntax:

$$\text{Inductive } id : A := c_1 : T \mid \dots \mid c_n : T_n$$

This defines id to be the smallest type in A closed under the constructors c_1, \dots, c_n .

Example 1. The type of natural numbers is defined as follows:

```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat.
```

We make a systematic use of the proof principle of *inductive inversion* (see [7] for a detailed account of its implementation) that applies to proofs of inductive terms and allows case-based reasoning.

Example 2. In this simple example we derive the *inversion statement* SS_{nm} below by applying the inversion proof principle to the hypothesis $S\ n \leq S\ (S\ m)$ and performing case analysis using the additional hypotheses generated by this application.

```
Inductive le (n : nat) : nat -> Prop :=
  | le_n : n <= n
  | le_S : forall m : nat, n <= m -> n <= S m.
```

```
Lemma SSnm : forall n m,
  S n <= S (S m) -> n <= S m.
intros n m H.
inversion H.
```

The goal here splits in two. The first subgoal is $S\ m \leq S\ m$ with an additional assumption $n = S\ m$. This subgoal is trivially proved by a constructor of `le`.

```
constructor.
```

The last subgoal is $n \leq S\ m$ with additional assumptions $S\ n \leq S\ m$, $m0 : nat$ and $m0 = S\ m$. This subgoal is proved by transitivity of `le`, constructors of `le`, and the assumption $S\ n \leq S\ m$.

```
apply le_trans with (S n).
repeat constructor.
assumption.
```

```
Qed.
```

We also make use of the sectioning discipline in Coq. That is we organise definitions and proofs in sections, each with a separate set of variables and hypotheses. This allows us to group proofs with respect to their formal domain. In fact, Sections 3–6, which are merely transcripts of the formal proof code, have separate environments. Moreover, within these environments we use an ordering of variables and hypotheses, which allows to saturate the section environment in a sequential way. Therefore those theorems that appear towards the end of a section would generally have larger environments comparing to those appearing at the beginning.

We represent finite sets by plain lists. Although we sometimes we use the notation with braces to denote sets, this is merely made for typographic reasons rather than to abstract from the implementation of sets. Lists are a sufficient setting for the regular expression theorem and no special set-theoretic library is required. Set theoretic operations on lists are defined using the membership relation. Therefore neither multiplicities of elements in a list nor duplications concern us.

3 Languages

Until the end of this section, assume we are given an arbitrary type A which represents the universe of symbols. A *word* over A is defined to be a list of type A . In effect, the type of words is essentially the type of polymorphic lists. We will omit the implicit parameter A when we talk about words in this section.

A *language* over A is a proposition-valued function from words. Similarly to words, we will omit A when talking about languages. We say that a language l *contains* a word w if $l\ w$ holds. Any two

languages l_1 and l_2 are said to be *equivalent*, denoted $l_1 \simeq l_2$, if they contain the same words. It is easy to see that, since word containment is undecidable, language equivalence is also undecidable.

Elementary languages are defined as follows. The *empty language* l_0 is the function that is constantly false. A *one-word language* $l_{\{w\}}$ is a function from words to languages mapping a word w_0 to the equality $w_0 = w$, for w a constant word. The *unit language* $l_{\{nil\}}$ is the one-word language for *nil*. A *one-symbol language* $l_{\{a::nil\}}$ is a one-word language for $a:A$.

Given a language l and a word w , the *left quotient* of l with respect to w , denoted $w \setminus l$, is defined inductively as the least language containing every word u such that $l(w++u)$ holds. Symmetrically, given a language l and a word u , the *right quotient* of l with respect to u is the least language containing every word w such that $l(w++u)$ holds.

Lemma 1. $w \setminus l_{\{w\}} \simeq l_{\{nil\}}$.

Proof. By inductive inversion on the premise of each conjunct of the language equivalence. \square

Operations on languages are defined by induction as follows. The *union* of two languages l_1 and l_2 , denoted $l_1 + l_2$, is the least language containing every word w such that $l_1 w$ or $l_2 w$. The *concatenation* of two languages l_1 and l_2 , denoted $l_1 \cdot l_2$, is the least language containing all words of the form w_1++w_2 such that $l_1 w_1$ and $l_2 w_2$. For a language l and a natural number n , the *n-th power* of l , denoted l^n , is defined recursively as follows:

$$l^n = \begin{cases} \text{if } n = 0 \text{ then } l_{\{nil\}} \\ \text{else if } n = S p \text{ then } l \cdot l^p \end{cases}$$

The *Kleene closure* of a language l , denoted l^* , is defined as the least language containing the n -th power of l for any n .

Union, concatenation and Kleene closure are a sufficient setting for the regular expression theorem. For a better expressivity we may also consider the following operators. The *intersection* of two languages l_1 and l_2 , denoted $l_1 \cap l_2$, is the least language containing every word w such that both $l_1 w$ and $l_2 w$ hold. The *complement* of a language l , denoted $\neg l$, is the greatest language containing every word w such that $l w$ does not hold.

A *regular expression*, over a finite list of type A , denoting a language is a function from lists of A to languages to types that is defined by induction as follows. For any list X of A , the *zero regular expression* over X denotes l_0 . For any X , the *unit regular expression* over X denotes $l_{\{nil\}}$. For any X and $a:A$, if $a \in X$, the *one-symbol regular expression* of a denotes $l_{\{a::nil\}}$. For any X and two languages l_1 and l_2 , given a regular expression over X denoting l_1 and a regular expression over X denoting l_2 , the *union regular expression* over X denotes $l_1 + l_2$. For any X and two languages l_1 and l_2 , given a regular expression over X denoting l_1 and a regular expression over X denoting l_2 , the *concatenation regular expression* over X denotes $l_1 \cdot l_2$. For any X and l , given a regular expression over X denoting l , the *Kleene closure regular expression* over X denotes l^* . Nothing else is a regular expression.

Note that the type of regular expressions depends on the denoted language and therefore follows the definition in [13], in contrast with the traditional definition [11, 12] that makes no implicit connection with underlying languages. For comparison, this dependency allows to embed the language meaning in a regular expression in a type-theoretic style (rather than to introduce a separate matching semantics later on) and to inspect this meaning using inductive constructors of the type of regular expressions. A similar dependent implementation is found in [9]. The type of languages is found across a number of proof assistants, among these is Nuprl [5].

4 Non-deterministic finite automata

Until the end of this section, assume we are given a type *Symbol* of *symbols*, and a finite list Σ of elements of type *Symbol* that will represent the *alphabet*. Assume also a type *State* of *states*, a finite list Q of states, an *initial state* q_I , a finite list F of *final states*, and a *transition relation* δ which is a finite list of triples of type $State \times Symbol \times State$.

A *non-deterministic finite automaton* (NFA) is an inductively defined proposition with the following premises: $q_I \in Q$, $F \subseteq Q$, $\delta \subseteq \Sigma \times Q \times Q$. Therefore, to say that $(\Sigma, Q, q_I, F, \delta)$ is an NFA, is equivalent to saying that the NFA proposition holds for these five.

A *path* from a state to a state of an NFA is a language of type *Symbol* defined inductively as follows. For all $q \in Q$, the *reflexivity path* from q to q is the smallest language containing *nil*, that is $l_{\{nil\}}$. For all q_1, q_2 and w such that there is a path from q_1 to q_2 containing w , for all $a \in \Sigma$ and $q \in Q$ such that $(q, a, q_1) \in \delta$, the *transitivity path* from s to q_2 is the smallest language containing $a::w$. Nothing else is a path. Assuming transitions δ , we will write $q_1 \xrightarrow{w} q_2$ to mean that w is contained in the path from q_1 to q_2 defined over δ .

The *accepted language of an NFA* is an inductively defined language of *Symbol* parametrised by the premises, for all w and $q_F, q_I \in F$ and $q_I \xrightarrow{w} q_F$.

Finally, the inductive type of *regular languages* is parametrised by a language of *Symbol*. In other words, it is an inductive informative predicate of one argument (in the present environment) whose type is a language of *Symbol*. It is defined by a single inductive clause that, for all l , if the NFA proposition holds for Σ, Q, q_I, F and δ , and if $l \simeq l_0$ holds for the language l_0 accepted by the automaton then l is a regular language. In a given environment containing an alphabet, a list of states, an initial state, a list of final states and a list of transitions, saying that l is a regular language is equivalent by definition to saying that the informative regular language predicate holds for l .

5 NFA constructions

By analogy with Section 4, assume until the end of this section types *Symbol* and *State* of symbols and states, respectively.

Lemma 2. *For any finite list Σ of *Symbol*, and states q_I and q_F such that $q_I \neq q_F$, the regular language predicate holds for Σ , $Q = \{q_I, q_F\}$, q_I , $F = \{q_F\}$, $\delta = \emptyset$ and the empty language of *Symbol*.*

Proof. It is required to prove the premises of the informative regular language predicate. The NFA proposition holds trivially. For any word w of *Symbol*, the “if” direction of the language equivalence is proved by contradiction, and the “only if” direction is proved by inductive inversion on the antecedent of the implication – that is the inductively defined statement that w is contained in the language accepted by the automaton, – which derives assumptions $q'_F : State$, $H^\epsilon : q'_F \in \{q_F\}$ and $H^\rightarrow : q_I \xrightarrow{w} q'_F$. The proof is essentially concluded by inductive inversion on the assumption H^\rightarrow . \square

Lemma 3. *For any finite list Σ of *Symbol*, and a state q_I , the regular language predicate holds for Σ , $Q = \{q_I\}$, q_I , $F = \{q_I\}$, $\delta = \emptyset$ and the unit language of *Symbol*.*

Proof. The proof is a slight modification of the proof of Lemma 2. The difference is that, for any w , the “if” direction of the language equivalence is (more generally) proved by inductive inversion on the antecedent of the implication, namely $l_{\{nil\}} w$. \square

There is a great deal of similarity on the structural level between the proof of Lemma 3 and the more involved proofs below in this section.

Lemma 4. For any finite list Σ of Symbol, states q_I and q_F , and a symbol a satisfying $q_I \neq q_F$ and $a \in \Sigma$, the regular language predicate holds for Σ , $Q = \{q_I, q_F\}$, q_I , $F = \{q_F\}$, $\delta = \{(q_I, a, q_F)\}$ and the one-symbol language of a (of type Symbol).

Proof. The proof follows the guidelines of the proof of Lemma 3. When proving the “if” direction of the language equivalence for any w , it is required to prove that $a::nil$ is a transitivity path over δ . In the proof of the “only if” direction, for any w , application of inductive inversion to the antecedent generates assumptions $q'_F::State$, $H^\inleftarrow : q'_F \in \{q_F\}$ and $H^\rightarrow : q_I \xrightarrow{w} q'_F$. The proof then proceeds by a sequential application of inductive inversion to H^\rightarrow and H^\inleftarrow . \square

Assume for the rest of this section the *empty symbol* ε of type Symbol satisfying the following conditions:

$$\text{for any } \Sigma, \quad \varepsilon \in \Sigma \quad (1)$$

$$\text{for any } w, \quad \varepsilon::w = w \quad (2)$$

Lemma 5. Assume arbitrary finite list Σ of Symbol, finite lists of states Q_1 and Q_2 , states q_I^1, q_I^2 and q_I , finite lists of states F_1 and F_2 , transition relations δ_1 and δ_2 , and languages l_1 and l_2 satisfying $\neg(Q_1 \cap Q_2)$, $q_I \notin Q_1$, $q_I \notin Q_2$, and such that l_1 is a regular language of $(\Sigma, Q_1, q_I^1, F_1, \delta_1)$ and l_2 is a regular language of $(\Sigma, Q_2, q_I^2, F_2, \delta_2)$. Then $l_1 \cup l_2$ is a regular language of $(\Sigma, \{q_I\} \cup Q_1 \cup Q_2, q_I, F_1 \cup F_2, \{(q_I, \varepsilon, q_I^1), (q_I, \varepsilon, q_I^2)\} \cup \delta_1 \cup \delta_2)$.

Proof. Follows the shape of the proof of Lemma 4. The crucial case analysis in the proof of the NFA proposition is done by inductive inversion on the derived hypothesis

$$H_t : t \in \{ \{(q_I, \varepsilon, q_I^1), (q_I, \varepsilon, q_I^2)\} \cup \delta_1 \cup \delta_2 \}$$

The “if” direction of the language equivalence is proved by properties of lists and an injectivity property of paths which is again derived from elementary properties of lists. The “only if” direction is proved using the fact that paths starting in Q_1 (respectively Q_2) can be restricted to the automaton $(\Sigma, Q_1, q_I^1, F_1, \delta_1)$ (respectively $(\Sigma, Q_2, q_I^2, F_2, \delta_2)$). \square

The proof of Lemma 6 below requires the following decidability assumption:

$$\text{for any } a \text{ in Symbol, either } a = \varepsilon \text{ or } a \neq \varepsilon \text{ is provable} \quad (3)$$

This still does not require Symbol to be decidable.

Lemma 6. Assume arbitrary finite list Σ of Symbol, finite lists of states Q_1 and Q_2 , states q_I^1 and q_I^2 , finite lists of states F_1 and F_2 , transition relations $\delta_1, \delta_\curvearrowright$ and δ_2 , and languages l_1 and l_2 satisfying $\neg(Q_1 \cap Q_2)$, and such that, for all $(q_1, a, q_2) \in \delta_\curvearrowright$, it holds that $q_1 \in F_1$, $a = \varepsilon$ and $q_2 = q_I^2$; moreover, it holds that l_1 is a regular language of $(\Sigma, Q_1, q_I^1, F_1, \delta_1)$ and l_2 is a regular language of $(\Sigma, Q_2, q_I^2, F_2, \delta_2)$. Then $l_1 \cdot l_2$ is a regular language of $(\Sigma, Q_1 \cup Q_2, q_I, F_2, \delta_1 \cup \delta_\curvearrowright \cup \delta_2)$.

Proof. Follows the shape of the proof of Lemma 4. \square

Lemma 7. Assume arbitrary finite list Σ of Symbol, a finite list of states Q , states q_I and q_I^* , a finite list of states F , transition relations δ and δ_\curvearrowright , and a language l satisfying $q_I^* \notin Q$, and such that, for all $(q_1, a, q_2) \in \delta_\curvearrowright$, it holds that $q_1 \in F$, $a = \varepsilon$ and $q_2 = q_I^*$; moreover, it holds that l is a regular language of $(\Sigma, Q, q_I, F, \delta)$. Then l^* is a regular language of $(\Sigma, \{q_I^*\} \cup Q, q_I^*, \{q_I^*\}, \{(q_I^*, \varepsilon, q_I)\} \cup \delta \cup \delta_\curvearrowright)$.

Proof. Follows the shape of the proof of Lemma 4. \square

In the following section we will employ a fixpoint function mk_δ that creates transitions from a given list of states to a given state and labels each of these transitions with a given symbol. This function is defined as follows:

$$\begin{aligned} \text{fix } mk_\delta (Q : \text{list State}) (q : \text{State}) (a : \text{Symbol}) := \\ \text{match } Q \text{ with } \quad \text{nil} \Rightarrow \text{nil} \mid q_0 :: Q_0 \Rightarrow (q_0, a, q) :: mk_\delta Q_0 q a \end{aligned} \quad (4)$$

The list of transitions resulting from application of mk_δ to Q , s and a will be denoted as $Q \xrightarrow{a} q$.

6 From regular expressions to NFAs

Assume a type *Symbol* and a finite list Σ of *Symbol*. Assume the empty symbol of *Symbol* satisfying (1)–(2).

We introduce a few helper functions to work with sequences of natural numbers that occur in the proof of the theorem.

The *lift function* that increments every element of a given sequence of natural numbers by a given natural number is defined as follows:

$$\lambda n : \text{nat}. \text{map } (\lambda s : \text{nat}. s + n) \quad (5)$$

We will use the notation $Q \uparrow^n$ to denote the sequence that is the result of lifting the sequence Q by the number n .

To account for lists of transitions, we should extend the above definition of the lift function in the following way. The *lift function on transitions* that increments both source and destination of each transition in a given list of transitions by a given number has the following definition:

$$\lambda n \delta. \text{map } (\lambda t. \text{let } (q_1, a, q_2) = t \text{ in } (q_1 + n, a, q_2 + n)) \delta \quad (6)$$

We will use the notation $\delta \uparrow^n$ to denote the list that is the result of lifting the list of transitions δ by the number n .

Theorem 8. *For any language l of *Symbol* and a regular expression r of Σ denoting l , one can construct a finite sequence Q of natural numbers, or states, from 0 to $|Q| - 1$, a state q_I , a finite list of states F and a transition relation δ of type $\text{nat} \times \text{Symbol} \times \text{nat}$ such that l is a regular language of $(\Sigma, Q, q_I, F, \delta)$.*

Proof. By induction on r . There are three basic cases:

If r is the zero regular expression, take $Q = 0 :: 1 :: \text{nil}$, $q_I = 0$, $F = \{1\}$ and $\delta = \emptyset$, and apply Lemma 2.

If r is the unit regular expression, take $Q = 0 :: \text{nil}$, $q_I = 0$, $F = \{0\}$ and $\delta = \emptyset$, and apply Lemma 3.

For any $a \in \Sigma$, if r is the one-symbol regular expression of a then take $Q = 0 :: 1 :: \text{nil}$, $q_I = 0$, $F = \{1\}$ and $\delta = \{(0, a, 1)\}$, and apply Lemma 4.

There are three cases to consider on the inductive step:

If r is the union of some regular expressions r_1 and r_2 then, by inductive hypothesis, r_1 denotes a regular language l_1 of an NFA $(\Sigma, Q_1, q_I^1, F_1, \delta_1)$ and r_2 denotes a regular language l_2 of an NFA $(\Sigma, Q_2, q_I^2, F_2, \delta_2)$. Then take $Q = 0 :: Q_1 \uparrow^1 ++ Q_2 \uparrow^{|Q_1|+1}$, $q_I = 0$, $F = F_1 \uparrow^1 \cup F_2 \uparrow^{|Q_1|+1}$ and $\delta = \{(0, \varepsilon, q_I^1 + 1), (0, \varepsilon, q_I^2 + |Q_1| + 1)\} \cup \delta_1 \uparrow^1 \cup \delta_2 \uparrow^{|Q_1|+1}$, and apply Lemma 5.

If r is the concatenation of some regular expressions r_1 and r_2 then, by inductive hypothesis, r_1 denotes a regular language l_1 of an NFA $(\Sigma, Q_1, q_I^1, F_1, \delta_1)$ and r_2 denotes a regular language l_2 of

an NFA $(\Sigma, Q_2, q_I^2, F_2, \delta_2)$. Then take $Q = Q_1 ++ Q_2 \uparrow^{|Q_1|}$, $q_I = q_I^1$, $F = F_2 \uparrow^{|Q_1|}$ and $\delta = \delta_1 \cup (F_1 \xrightarrow{\varepsilon} (q_I^2 + |Q_1|)) \cup \delta_2 \uparrow^{|Q_1|}$, and apply Lemma 6.

If r is the Kleene closure of some regular expression r_0 then, by inductive hypothesis, r_0 denotes a regular language l_0 of an NFA $(\Sigma, Q_0, q_I^0, F_0, \delta_0)$. Then take $Q = 0 :: Q_0 \uparrow^1$, $q_I = 0$, $F = \{0\}$ and $\{(0, \varepsilon, q_I + 1)\} \cup \delta_0 \uparrow^1 \cup (F_0 \uparrow^1 \xrightarrow{\varepsilon} 0)$, and apply Lemma 7. \square

7 Extraction of a functional program

The type of the main theorem in Coq is the following:

```

regexp_nf_automaton :
  forall (Symbol:Type)
    (symbols:list Symbol)
    (empty_symbol:Symbol),
  (forall symbols0:list Symbol,
    In empty_symbol symbols0) ->
  (forall w:list Symbol,
    empty_symbol::w = w) ->
  (forall a b:Symbol, {a = b} + {a <> b}) ->
  forall l:language Symbol,
    regexp symbols l ->
    {states : list State &
     states = seq 0 (length states) &
     {init_state : State &
      {acc_states : list State &
       {transition : list (State * Symbol * State) &
        nf_regular_language symbols states init_state
         acc_states transition l}}}}

```

To extract it to Caml, we use the standard Coq command `Extraction`. The type of the extracted Caml program is more succinct because the extraction mechanism discarded non-informative parameters:

```

regexp_nf_automaton :
  'a list -> 'a -> ('a -> 'a -> bool) -> 'a regexp ->
  (nat list, Obj.t,
   nat * (nat list *
    (((nat * 'a) * nat) list *
     ('a, nat) nf_regular_language)))) sigT2

```

Thus there are only four parameters left now: the list of symbols, the empty symbol, the decider function for the type of symbols, and a regular expression parametrised by the type of symbols. The result is the automaton constructed in the proof of Theorem 8.

Example 3. Take the regular expression $K1(e*)ne$. We will apply the extracted function in Caml toplevel to obtain an automaton accepting the language denoted by this regular expression. The decider function for the type `char` is defined as follows:

```

# type char_dec = char -> char -> bool;;

# let char_dec =

```

```

fun c1 c2 ->
  if c1 = c2 then true else false;;

```

We define the set of symbols and encode the regular expression, and then apply the main function to these arguments taking `_` as the empty character.

```

# let symbols = ['K'; 'l'; 'e'; 'n'];;

# let r = Regexp_cat (symbols,
  (Regexp_cat (symbols,
    (Regexp_cat (symbols, (Regexp_sym (symbols, 'K')),
      (Regexp_sym (symbols, 'l')))),
    (Regexp_closure (symbols,
      (Regexp_sym (symbols, 'e'))))),
  (Regexp_cat (symbols, (Regexp_sym (symbols, 'n')),
    (Regexp_sym (symbols, 'e')))));;

# regexp_nf_automaton symbols '_' char_dec r;;

```

The resulting automaton has states 0 through 10, with 0 being the initial state and 10 being the only final state. The list of transitions is the following:

```

[(0, 'K', 1); (1, '_', 2); (2, 'l', 3); (3, '_', 4);
 (4, '_', 5); (5, 'e', 6); (6, '_', 4); (4, '_', 7);
 (7, 'n', 8); (8, '_', 9); (9, 'e', 10)]

```

An example of a string that is accepted by this automaton is `K_l__e__e__n_e` which, by the property (2), is equivalent to Kleene.

The obvious downside of the present method is that in the example above more than a half of transitions are the idle ones. The method uses the empty symbol as a proof convenience which is however responsible for production of computationally cumbersome automata.

8 Further directions

We are developing an alternative library that does not make use of the sometimes problematic empty symbol and the associated axioms. We found it possible to adapt the proofs presented in this paper to the kind of automata constructions known as Glushkov automata [10] (also discussed in [13]). Glushkov automata merge transitions instead of introducing new intermediate states connected by idle transitions. However, our proofs with the empty symbol seem to be a little bit less involved when arithmetic is concerned. So, in this respect they might be regarded as a step towards the proofs without the empty symbol.

Another direction of development is marked by the seminal paper [1] that introduced a concept of a partial derivative of a regular expression (see also [4]). This is the most interesting direction of research if dependent type theory is concerned. In this paper we use the dependent type of regular expressions. If we use structures such as left quotient languages from Section 3, as we do all the time with partial derivatives, we arrive at a considerably higher level of type dependency. In Section 3, we already mentioned that there are basically two definitions of regular expressions: the dependent and the non-dependent ones. The paper [1] uses the non-dependent definition because structural induction and termination was not a concern there. From the type theoretic point of view the lack of a dependently typed definition means there is more research to be done about the languages denoted by partial derivatives.

We also work on extensions of our library encompassing practical regular expressions [2, 3], that is regular expressions with additional operators such as backreferences that are standardised in operating system command shells and programming languages.

References

- [1] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
- [2] C. Campeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14:1007 – 1018, 2003.
- [3] B. Carle, P. Narendran, and C. Scheriff. On extended regular expressions. <http://hal.inria.fr/inria-00176043/en/>, 2007. HAL-CCSD preprint.
- [4] J.-M. Champarnaud and D. Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theor. Comput. Sci.*, 289(1):137–163, 2002.
- [5] R. L. Constable, P. B. Jackson, P. Naumov, and J. Uribe. Constructively formalizing automata theory. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 213–238. MIT Press, Cambridge, MA, USA, 2000.
- [6] Coq proof assistant. <http://coq.inria.fr/>.
- [7] C. Cornes and D. Terrasse. Automating inversion predicates in Coq. In *Workshop on Types for Proofs and Programs*, volume 1152 of *LNCS*, pages 85–104, 1995.
- [8] M. Day and V. Komendantsky. Coq proof developments of regular expressions, their partial derivatives, and NFAs. <http://similare.org/regexp-nfa.tgz>.
- [9] J.-C. Filliâtre. Finite automata theory in Coq – A constructive proof of Kleene’s theorem. Technical Report 97–04, Ecole Normale Supérieure de Lyon, CNRS UMR 1398, February 1997.
- [10] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(1), 1961.
- [11] D. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer, 1997.
- [12] M. D. McIlroy. Enumerating the strings of regular languages. *Journal of Functional Programming*, 14:503–518, 2004.
- [13] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, volume 1: Word, language, grammar, pages 41–110. Springer-Verlag, 1997.

A The extracted auxiliary function definitions

```
type __ = Obj.t
let __ = let rec f _ = Obj.repr f in Obj.repr f

type nat = 0 | S of nat

type ('a, 'p, 'q) sigT2 = ExistT2 of 'a * 'p * 'q

let rec plus n m =
  match n with
```

```

    | 0 -> m
    | S p -> S (plus p m)

let rec length = function
  | [] -> 0
  | a :: m -> S (length m)

let rec app l m =
  match l with
  | [] -> m
  | a :: l1 -> a :: (app l1 m)

let rec map f = function
  | [] -> []
  | a :: t -> (f a) :: (map f t)

let lift n l = map (fun s -> plus s n) l

type 'a regexp =
  | Regexp_empty of 'a list
  | Regexp_unit of 'a list
  | Regexp_sym of 'a list * 'a
  | Regexp_union of 'a list * 'a regexp * 'a regexp
  | Regexp_cat of 'a list * 'a regexp * 'a regexp
  | Regexp_closure of 'a list * 'a regexp

type ('symbol, 'state) nf_regular_language = Nf_regular_language

let regexp_nf_automaton_union empty_symbol symbols states1 states2 init1 init2
  init_union acc1 acc2 tr1 tr2 hl1 hl2 =
  let Nf_regular_language = hl1 in
  let Nf_regular_language = hl2 in Nf_regular_language

let regexp_nf_automaton_cat empty_symbol empty_symbol_dec symbols states1
  states2 init1 init2 acc1 acc2 tr1 tr_junction tr2 hl1 hl2 =
  let Nf_regular_language = hl1 in
  let Nf_regular_language = hl2 in Nf_regular_language

type state = nat

let lift_transitions n transitions =
  map (fun t ->
    let y , s2 = t in let s1 , a = y in ((plus s1 n) , a) , (plus s2 n))
    transitions

let rec lift_automaton n symbols0 states init acc tr hl =
  match n with
  | 0 -> hl
  | S n0 -> let Nf_regular_language = hl in Nf_regular_language

```

```

let rec mk_transitions from_states to_state sym =
  match from_states with
  | [] -> []
  | s :: ss -> ((s , sym) , to_state) :: (mk_transitions ss to_state sym)

```

B The extracted main function

```

let rec regexp_nf_automaton symbols empty_symbol symbol_dec = function
| Regexp_empty x -> ExistT2 ((0 :: ((S 0) :: [])), __, (0 , (((S 0) :: []))
  , ([] , Nf_regular_language))))
| Regexp_unit x -> ExistT2 ((0 :: []), __, (0 , ((0 :: [])) , ([] ,
  Nf_regular_language))))
| Regexp_sym (x, a) -> ExistT2 ((0 :: ((S 0) :: [])), __, (0 , (((S 0) ::
  [])) , (((0 , a) , (S 0)) :: [])) , Nf_regular_language))))
| Regexp_union (x, r0, r1) ->
  let ExistT2 (states1, _, s) =
    regexp_nf_automaton x empty_symbol symbol_dec r0
  in let init1 , s0 = s in let acc1 , s1 = s0 in let tr1 , hl1 = s1 in
  let ExistT2 (states2, _, s2) =
    regexp_nf_automaton x empty_symbol symbol_dec r1
  in let init2 , s3 = s2 in let acc2 , s4 = s3 in let tr2 , hl2 = s4 in
  ExistT2 ((0 ::
  (app (lift (S 0) states1) (lift (plus (length states1) (S 0)) states2))),
  __, (0 ,
  ((app (lift (S 0) acc1) (lift (plus (length states1) (S 0)) acc2)) ,
  (((0 , empty_symbol) , (plus init1 (S 0))) :: ((0 , empty_symbol) ,
  (plus init2 (plus (length states1) (S 0)))) ::
  (app (lift_transitions (S 0) tr1)
  (lift_transitions (plus (length states1) (S 0)) tr2)))) ,
  (regexp_nf_automaton_union empty_symbol x (lift (S 0) states1)
  (lift (plus (length states1) (S 0)) states2)
  (plus init1 (S 0)) (plus init2 (plus (length states1) (S 0))) 0
  (lift (S 0) acc1) (lift (plus (length states1) (S 0)) acc2)
  (lift_transitions (S 0) tr1)
  (lift_transitions (plus (length states1) (S 0)) tr2)
  (lift_automaton (S 0) x states1 init1 acc1 tr1 hl1)
  (lift_automaton (plus (length states1) (S 0)) x states2 init2 acc2
  tr2 hl2))))))
| Regexp_cat (x, r0, r1) ->
  let ExistT2 (states1, _, s) =
    regexp_nf_automaton x empty_symbol symbol_dec r0
  in let init1 , s0 = s in let acc1 , s1 = s0 in let tr1 , hl1 = s1 in
  let ExistT2 (states2, _, s2) =
    regexp_nf_automaton x empty_symbol symbol_dec r1
  in let init2 , s3 = s2 in let acc2 , s4 = s3 in let tr2 , hl2 = s4 in
  ExistT2 ((app states1 (lift (length states1) states2)), __, (init1 ,
  (lift (length states1) acc2) ,
  (app tr1

```

```

      (app
        (mk_transitions acc1 (plus init2 (length states1)) empty_symbol)
        (lift_transitions (length states1) tr2))) ,
    (regexp_nf_automaton_cat empty_symbol (fun x0 -> symbol_dec x0 empty_symbol)
      x states1 (lift (length states1) states2) init1
      (plus init2 (length states1)) acc1 (lift (length states1) acc2) tr1
      (mk_transitions acc1 (plus init2 (length states1)) empty_symbol)
      (lift_transitions (length states1) tr2) hl1
      (lift_automaton (length states1) x states2 init2 acc2 tr2 hl2))))))
| Regexp_closure (x, r0) ->
  let ExistT2 (states, _, s) =
    regexp_nf_automaton x empty_symbol symbol_dec r0
  in let init , s0 = s in let acc , s1 = s0 in let tr , hl = s1 in
  ExistT2 ((0 :: (lift (S 0) states)), __, (0 , ((0 :: [])) , (((0 ,
  empty_symbol) , (plus init (S 0))) ::
  (app (lift_transitions (S 0) tr)
    (mk_transitions (lift (S 0) acc) 0 empty_symbol))) ,
  (let Nf_regular_language = lift_automaton (S 0) x states init acc tr hl
    in
  Nf_regular_language))))))

```