Partial Derivative Automata Formalized in Coq

José Bacelar Almeida[3] Nelma Moreira[1]
David Pereira[1] Simão Melo de Sousa[2]

[1] DCC-FC & LIACC, University of Porto
Rua do Campo Alegre 1021, 4169-007, Porto, Portugal
{nam,dpereira}@ncc.up.pt
[2] LIACC & DI, University of Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001 Covilhã, Portugal
desousa@di.ubi.pt
[3] CCTC & DI, University of Minho,
Campus de Gualtar, 4710-057 Braga,Portugal
jba@di.uminho.pt

Abstract. In this paper we present a computer assisted proof of the correctness of a partial derivative automata construction from a regular expression within the Coq proof assistant. This proof is part of a formalization of Kleene algebra and regular languages in Coq towards their usage in program certification.

1 Introduction

The use of proof assistants has gained increasing importance in mathematics and computer science. Their value in the assurance of theorem and algorithm correctness is obvious, since all the steps and intricacies involved in the proof process are formally and mechanically checked.

The use of the Coq proof assistant [BC04] for program verification is specially attractive because correctness proofs can be compiled as *proof certificates*, and the constructive components of the specification and proof development can be extracted into functional programs.

In this paper we describe a formalization of regular languages in Coq. Our main result is the proof of the correctness of a partial derivative automata construction from a regular expression. This result is a step towards the implementation of a proved terminating, and correct, decision procedure for regular expression equivalence based on the notion of (partial) derivatives. From such implementation it is possible to extract a *correct-by-construction* functional program, and it is also possible to develop *proof tactics* that automate the construction of proofs.

Kleene algebra can be used to capture several properties of programs. In this setting, testing Kleene algebra terms equivalence can correspond to proving partial correctness of programs. Defining and proving the correctness of a decision procedure within a proof assistant that features *proof objects*[4] allows to obtain certificates that facilitate the automations of formal software verification.

[4] In such systems proof objects are values that can be compiled into binary code.

The paper is organised as follows. In Section 2 we review some definitions about regular languages and finite automata. The partial derivative automaton and Mirkin's construction are reviewed in Section 3. Section 4 presents a small introduction to the Coq proof assistant. In Section 5 we describe the formalization of regular languages in Coq and present the main result. In Section 6 we comment on related work. Finally, in Section 7 we draw some conclusions and point some future work.

2 Regular Languages and Finite Automata

Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$ be an *alphabet* (set of *symbols*). A *word* w over Σ is any finite sequence of symbols. The *empty word* is denoted by ε. The *concatenation* of two words w_1 and w_2 is the word $w = w_1 w_2$. Let $\Sigma^\star$ be the set of all words over Σ. A *language* over Σ is a subset of $\Sigma^\star$. If L_1 and L_2 are two languages, then $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$. The operator $\cdot$ is often omitted. The *power of a language* L is inductively defined by $L^0 = \{\varepsilon\}$, $L^n = LL^{n-1}$, for $n \geq 1$. The Kleene *star* $L^\star$ of a language L is $\cup_{n \geq 0} L^n$. Given a word $w \in \Sigma^\star$ and $L \in \Sigma^\star$, the (*left-*)*quotient* of L by w is $w^{-1}L = \{v \mid wv \in L\}$.

A *regular expression* (re) α over Σ represents a *regular language* $\mathcal{L}(\alpha) \subseteq \Sigma^\star$ and is inductively defined by: $\emptyset$ is a re and $\mathcal{L}(\emptyset) = \emptyset$; ε is a re and $\mathcal{L}(\varepsilon) = \{\varepsilon\}$; $a \in \Sigma$ is a re and $\mathcal{L}(a) = \{a\}$; if α_1 and α_2 are re, $(\alpha_1 + \alpha_2)$, $(\alpha_1 \alpha_2)$ and $(\alpha_1)^\star$ are re, respectively with $\mathcal{L}((\alpha_1 + \alpha_2)) = \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2)$, $\mathcal{L}((\alpha_1 \alpha_2)) = \mathcal{L}(\alpha_1)\mathcal{L}(\alpha_2)$ and $\mathcal{L}((\alpha_1)^\star) = \mathcal{L}(\alpha_1)^\star$. The *alphabetic size* of an re α is the number of symbols of α and it is denoted by $|\alpha|_\Sigma$. The *constant part* of an re is denoted by $\varepsilon(\alpha)$ and defined by $\varepsilon(\alpha) = \varepsilon$ if $\varepsilon \in \mathcal{L}(\alpha)$, and $\varepsilon(\alpha) = \emptyset$ otherwise. The same function can be applied to languages. Let RE be the set of regular expressions over Σ. If two re's α and β are syntactically identical, we write $\alpha \equiv \beta$. Two re's are equivalent if they represent the same regular language, that is, if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, and we write $\alpha = \beta$. The equational properties of regular expressions are axiomatically captured by a *Kleene algebra* (KA), normally called *the algebra of regular events*, after the seminal work of S. C. Kleene [Kle56]. A KA is an algebraic structure $(K, 0, 1, +, \cdot, ^\star)$ such that $(K, 0, 1, +, \cdot)$ is an *idempotent semiring* and where the operator * (Kleene's *star*) is characterized by a set of axioms. The algebra of regular events is given by $(\mathsf{RE}, \emptyset, \varepsilon, +, \cdot, ^\star)$. There are several ways of axiomatizing a KA but we considered the one presented by D. Kozen in [Koz94].

A *non-deterministic finite automaton* (NFA) $\mathcal{A}$ is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, q_0 the initial state and $F \subseteq Q$ is the set of final states. For $q \in Q$ and $a \in \Sigma$, we denote the set $\{p \mid (q, a, p) \in \delta\}$ by $\delta(q, a)$, and we can extend this notation to $w \in \Sigma^\star$, and to $R \subseteq Q$. An NFA is *deterministic* (DFA) if for each pair $(q, a) \in Q \times \Sigma$, $|\delta(q, a)| \leq 1$. The *language recognized* by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\star \mid \delta(q_0, w) \cap F \neq \emptyset\}$. The set of languages recognized by NFA' s coincides with the set of languages represented by regular expressions, i.e the set of *regular languages*.

Regular languages can be associated to sets of languages equations. Given an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $|Q| = n + 1$ we can consider $Q = [0, n]$ and $q_0 = 0$. Let L_i be the language recognized by the automaton $([0, n], \Sigma, \delta, i, F)$, for $i \in [0, n]$ and $\mathcal{L}(A) = L_0$. Then, the following language equations are satisfied:

$$\begin{array}{ll} L_i &= \left(\bigcup_{j=1}^{k} \{a_j\} L_{ij}\right) \cup \varepsilon(L_i), \quad \forall i \in [0, n], \\ L_{ij} &= \bigcup_{m \in I_{ij}} L_m, \quad I_{ij} = \delta(i, a_j) \subseteq [0, n] \end{array} \tag{1}$$

Conversely any set of languages $\{L_0, \ldots, L_n\}$ that satisfies the set of equations (1) defines an NFA with initial state L_0. In particular if L_i are represented by regular expressions $\alpha \equiv \alpha_0, \ldots, \alpha_n$ the following set of equations holds:

$$\begin{array}{ll} \alpha &\equiv \alpha_0 \\ \alpha_i &= a_1\alpha_{i1} + \ldots + a_k\alpha_{ik} + \varepsilon(\alpha_i), \text{ for } i \in [0, n] \\ \alpha_{ij} &= \sum_{m \in I_{ij}} \alpha_m, \quad I_{ij} \subseteq [0, n] \end{array} \tag{2}$$

Given $\alpha \in \mathsf{RE}$, to find a set of re's that satisfies (2) is tantamount to find an NFA equivalent to α.

3 Partial Derivative Automata

There are several constructions to obtain NFA from re's. Based on the notion of derivative, Brzozowski [Brz64] established a construction of a DFA from a re. The partial derivative automaton ($\mathcal{A}_{pd}$), introduced by V. Antimirov [Ant96], is a non-deterministic version of the Brzozowski automaton.

For a re $\alpha \in \mathsf{RE}$ and a symbol $a \in \Sigma$, the set $\partial_a(\alpha)$ of *partial derivatives* of α w.r.t. a is defined inductively as follows:

$$\begin{array}{ll} \partial_a(\emptyset) = \partial_a(\varepsilon) = \emptyset & \partial_a(\alpha + \beta) = \partial_a(\alpha) \cup \partial_a(\beta) \\ \partial_a(b) = \begin{cases} \{\varepsilon\} & \text{if } b \equiv a \\ \emptyset & \text{otherwise} \end{cases} & \partial_a(\alpha\beta) = \begin{cases} \partial_a(\alpha) \odot \beta \cup \partial_a(\beta) & \text{if } \varepsilon(\alpha) = \varepsilon \\ \partial_a(\alpha) \odot \beta & \text{otherwise,} \end{cases} \\ \partial_a(\alpha^\star) = \partial_a(\alpha) \odot \alpha^\star & \end{array}$$

where the operator $\odot$ is defined as follows. Let $S \subseteq \mathsf{RE}$ and $\beta \in \mathsf{RE}$. Then $S \odot \beta = \{\alpha\beta \mid \alpha \in S\}$ if $\beta \neq \emptyset$, and $S \odot \emptyset = \emptyset$ otherwise. Analogously, one defines $\beta \odot S$. Moreover if $S = \{\alpha_1, \ldots, \alpha_n\}$, let $\sum S$ denote the re $\alpha_1 + \cdots + \alpha_n$.

Lemma 1 (Antimirov). *For any $a \in \Sigma$ and $\alpha \in RE$, $\mathcal{L}(\sum \partial_a(\alpha)) = a^{-1}\mathcal{L}(\alpha)$.*

The definition of partial derivative can be extended to sets of regular expressions, words, and languages. Given $\alpha \in \mathsf{RE}$ and $a \in \Sigma$, $\partial_a(S) = \cup_{\alpha \in S} \partial_a(\alpha)$ for $S \subseteq \mathsf{RE}$, $\partial_\varepsilon(\alpha) = \{\alpha\}$, $\partial_{wa}(\alpha) = \partial_a(\partial_w(\alpha))$ for $w \in \Sigma^\star$, and $\partial_L(\alpha) = \cup_{w \in L} \partial_w(\alpha)$ for $L \subseteq \Sigma^\star$. Lemma 1 can be extended to words $w \in \Sigma^\star$. The *set of partial derivatives* of α is defined by $PD(\alpha) = \partial_{\Sigma^\star}\alpha$. An important fact is that $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$. Given a regular expression α, the *partial derivative automaton $\mathcal{A}_{pd}(\alpha)$* is thus defined as

$$\mathcal{A}_{pd}(\alpha) = (PD(\alpha), \Sigma, \delta_{pd}, \alpha, \{q \in PD(\alpha) \mid \varepsilon(q) = \varepsilon\}),$$

where $\delta_{pd}(q, a) = \partial_a(q)$, for all $q \in PD(\alpha)$ and $a \in \Sigma$.

Proposition 1 (Antimirov). $\mathcal{L}(\mathcal{A}_{pd}(\alpha)) = \mathcal{L}(\alpha)$.

Champarnaud and Ziadi [CZ01] proved that partial derivatives and Mirkin's prebases [Mir66] lead to identical constructions of non-deterministic automata. We now review Mirkin's construction. Given $\alpha \equiv \alpha_0 \in \mathsf{RE}$, the set $\pi(\alpha) = \{\alpha_1, \ldots, \alpha_n\}$, where $\alpha_1, \ldots, \alpha_n$ are non-empty re's, is called a *support* of α if it satisfies the set of equations (2), where α_{ij}, for $i \in [0, n]$ and $j \in [1, k]$, is a summation of elements of $\pi(\alpha)$. If $\pi(\alpha)$ is a support of α, then the set $\pi(\alpha) \cup \{\alpha\}$ is called a *prebase* of α. B. Mirkin provided an algorithm for the computation of a support of a re for which Champarnaud and Ziadi gave an elegant inductive definition[5].

Proposition 2 (Mirkin/Champarnaud&Ziadi). *Let $\alpha \in \mathsf{RE}$. Then, the set $\pi(\alpha)$, inductively defined by*

$$\begin{array}{ll} \pi(\emptyset) = \emptyset & \pi(\alpha + \beta) = \pi(\alpha) \cup \pi(\beta) \\ \pi(\varepsilon) = \emptyset & \pi(\alpha\beta) = \pi(\alpha) \odot \beta \cup \pi(\beta) \\ \pi(a) = \{\varepsilon\} & \pi(\alpha^\star) = \pi(\alpha) \odot \alpha^\star, \end{array}$$

is a support of α.

In his original paper Mirkin showed that $|\pi(\alpha)| \leq |\alpha|_\Sigma$. Furthermore, Champarnaud and Ziadi established that $PD(\alpha) = \pi(\alpha) \cup \{\alpha\}$. This fact can be proved noticing that $\mathcal{A}_{pd}(\alpha)$ verifies equations (1) which lead exactly to a language based version of equalities (2) when considering $\alpha_{ij} = \sum \partial_{a_j} \alpha_i$, for $i \in [0, n]$ and $j \in [1, k]$. To prove Proposition 1 is then equivalent to prove Proposition 2. The main result presented in this paper is the formalization of Proposition 2 in Coq.

4 The Coq Proof Assistant

The Coq proof assistant is an implementation of the *Calculus of Inductive Constructions* (CIC) [PM93], a typed λ-calculus that features polymorphism, dependent types and very expressive (co-)inductive types. Coq provides users with the means to define data-structures and functions, as in standard functional languages, and also allows to define specifications and to build proofs in the same language, if we consider the underlying λ-calculus as an higher-order logic under the *Curry-Horward isomorphism* programs-as-proofs principle (CHi) [SU98,How80].

In CHi, any typing relation $t : A$ can either be seen as a value t of type A, or as t being a proof of the proposition A. Any type in Coq is in the set of *sorts* $\mathcal{S} = \{\mathsf{Prop}\} \cup \{\mathrm{Type}(i) \mid i \in \mathbb{N}\}$. The Type(0) sort represents computational types, while the Prop type represents logical propositions.

An inductive type is introduced by a collection of *constructors*, each with its own arity. A value of an inductive type is a composition of such constructors. As an example, natural numbers are encoded as follows:

```
Inductive ℕ : Type :=
| 0 : ℕ | S : ℕ → ℕ.
```

[5] That definition was corrected by Broda *et al.* [BMMR].

Coq automatically generates induction and recursion principles for each new inductive type. More complex type families can be defined by combining inductive constructions and dependent types in Coq. We now introduce the *subset types* since they are used further ahead in this paper.

A *subset type* (or Σ-type) is a dependent type that combines datatypes with predicates over these types, thus determining a subset of the original datatype. In Coq, a subset type is defined as

```
Inductive sig (A:Type) (P:A → Prop) : Type :=
| exist : ∀ x:A, P x → sig P.
```

and has the special notation {x:A | P}. The constructor exist takes two arguments: a value x of type A and a term of type P(x) which is a proof that x verifies the logical properties of P. This last argument is usually called a *certificate*. Coq also provides *nested subset types* through the type

```
Inductive sigS (A:Type) (P:A → Type) : Type :=
| existS : ∀ x:A, P x → sigS A P.
```

which can be denoted {x:A & P}, with P being either a sig or a sigS type. This type permits one to consider the value of x from which we build values y such that Q(x,y) holds, for a given predicate Q, thus resulting in certified pairs/tuples of values.

In Coq, functions must be provably terminating. Termination is ensured by a guard predicate that checks that recursive calls are always performed on structurally smaller arguments. As an example, consider the function plus that adds two natural numbers.

```
Fixpoint plus (n m:ℕ) {struct n} : ℕ :=
 match n with
 | 0 ⇒ m | S p ⇒ S (plus p m)
 end.
```

The basic way of the Coq proof construction process is to explicitly build CIC terms. However, proofs can be built more conveniently and interactively in a backward fashion. This step by step process is done by the use of *proof tactics*.

Another appealing feature of Coq is the possibility to extract the constructive parts of proof development into correct by construction functional programs. Since the underlying logic of Coq is constructive, any value, proof included, can be seen as a (functional) program. The extraction mechanism keeps the computational counterparts and translate them into standard funcional programs. On the other hand, purely logical subterms are discarded since they are computationally non-informative.

Our formalization uses Coq *module system*, which allows to define both *module types*, and the usual notion of *modules*. A module type is a *signature* of a theory, that specifies its parameters and axioms. In an implementation of a module type, computational interpretations must be provided for parameters, and proofs must be given that assert the validity of the specified axioms. Modules are collections of components that form an implementation of a theory.

In this paper we use the Coq libraries Ensembles and FSets that formalize sets. The Ensembles library formalizes the notion of set as a characteristic predicate. The base type is Ensemble (X:Type) := X → Prop. Set operations are also provided. As an example, consider the *singleton* and the *union*:

```
Definition In(U:Type)(P:Ensemble U)(x:U) := P x.

Inductive Singleton(U:Type)(x:U) : Ensemble U :=
| In_singleton : In U (Singleton x) x.

Inductive Union(U:Type)(B C:Ensemble U) : Ensemble U :=
| Union_introl : ∀ x:U, In U B x → In U (Union B C) x
| Union_intror : ∀ x:U, In U C x → In U (Union B C) x.
```

The FSets library provides a rich implementation of finite sets over decidable and/or ordered types.

5 Formalization in Coq

This section describes the main parts of our formalization in Coq. First we present the formalization of regular languages and re's.

5.1 Formal Languages and Regular Expressions

An alphabet sigma (Σ) can be specified as a non-empty list of symbols of a type A. We also require that the type A is ordered, using the type Compare for defining the axiom compare_sy, that ensures that any two elements of type A can be compared.

```
Inductive Compare (A : Type) (lt eq : A → A → Prop) (x y : A) : Type :=
| LT : lt x y → Compare lt eq x y
| EQ : eq x y → Compare lt eq x y
| GT : lt y x → Compare lt eq x y.

Module Type Alphabet.
 Parameter A : Set.
 Definition A_eq := (eq A).
 Parameter A_lt : A → A → Prop.
 Parameter sigma : list A.
 Axiom sigma_nempty : sigma ≠ nil.
 Axiom compare_sy : ∀ x y:A, Compare sylt syeq x y.
End Alphabet.
```

Words are lists whose elements have type A, and that belong to sigma. A word w is a valid word if $w \in \Sigma^\star$ which correspond to the IsWord predicate.

```
Definition IsSy(a:A) := a ∈ sigma.
Definition word := list A.

Inductive IsWord : word → Prop :=
| nil_IsWord  : IsWord ε
| cons_IsWord : ∀ a:A, IsSy a → ∀ u:word, IsWord u → IsWord (a::u).
```

Languages are sets of words, that is, terms of type Ensemble word. The languages $\emptyset$, $\{\epsilon\}$, $\{a\}$ for $a \in \Sigma$, and language union are defined using the corresponding Ensembles definitions. Concatenation and Kleene's star are formalized as the predicates $\cdot$ and * as presented below. Equivalence of languages is denoted by $=_\mathcal{L}$ which is the standard set equivalence, and is represented by the predicate Same_set.

```
Definition language := Ensemble word.

Definition ∅ := (Empty word).
Definition ε := (Singleton word nil).
Definition ([S] x) := (Singleton word (x::nil)).
Definition (x ∪ y) := Union word x y.

Inductive (L1 · L2:language) : language :=
 | ConcL_app : ∀ w1 w2:word, w1 ∈ L1 → w2 ∈ L2 → (w1 ++ w2) ∈ (L1 · L2).

Fixpoint lpow (L:language)(n:ℕ) : language :=
 match n with
 |0 => ε  | (S m) => (L · (lpow L m))
 end.

Inductive (L:language)* : language :=
| starL_n : ∀ n:ℕ w, w ∈(lpow L n) → w ∈(L*).

Definition (L1 =ℒ L2:language) := (Same_set L1 L2).
```

Several properties of regular languages were proved, and, in particular, that regular languages are a model for KA. This was accomplished considering the KA implementation in Coq described in [PM08]. An extended description of that proof is presented in [MPdS09].

Regular expressions are encoded by the inductive type re. The language of any re α is obtained by applying the function re2rel to α. This function was proved correct w.r.t. to RL, the predicate that defines regular languages over the alphabet sigma (Theorem re2rel_is_RL).

```
Inductive re : Set :=
| re0 : re | re1 : re | re_sy : ∀ a:A, IsSy a → re
| re_union : re → re → re | re_conc : re → re → re
| re_star : re → re.

Fixpoint re2rel (α:re) : language :=
 match x with
 | re0 ⇒ ∅ | re1 ⇒ ε | re_sy a H ⇒ ([S] a)
 | re_union α1 α2 ⇒ (re2rel α1) ∪ (re2rel α2)
 | re_conc α1 α2 ⇒ (re2rel α1) · (re2rel α2)
 | re_star α1 ⇒ (re2rel α1) [∗]
 end.

Coercion re2rel : re ↣ language.

Inductive RL : language → Prop :=
| RL0 : RL ∅ | RL1 : RL ε | RLs : ∀ a, RL ([S] a)
| RLp : ∀ l1 l2, RL l1 → RL l2 → RL (l1 ∪ l2)
| RLc : ∀ l1 l2, RL l1 → RL l2 → RL (l1 · l2)
| RLst : ∀ l, RL l → RL (l*).

Theorem re2rel_is_RL : ∀ α:re, RL α.
```

In the code above, re2rel was declared as a *coercion* which allows one to refer to a given re α where its language is required, i.e., without explicitly referring to re2rel(α). For instance, in the case of the concatenation of the languages corresponding to the re's α_1 and α_2, we write $\alpha_1 \cdot \alpha_2$ instead of (re2rel(α_1))·(re2rel (α_2)).

5.2 Correctness of Mirkin's Construction

We now present the formalization of our main result, i.e., given a re α, the function $\pi(\alpha)$ computes a support for α.

The support π is formalized in Coq as a structural recursive function, and thus its termination is ensured. The function $\_ \odot \_$ is defined using the auxiliary function fold_conc, which concatenates a re to the right of each element of a set of re's. A set of re's is represented by the type re_set).

```
Definition fold_conc (s:re_set)(r:re) :=
  fold (fun x ⇒ add (re_conc x r)) s empty.

Definition ⊙ (s:re_set)(r:re):t :=
  match r with
  | re0 ⇒ empty | _ ⇒ fold_conc s r
  end.

Fixpoint π (r:re) : re_set :=
 match r with
 | re0 ⇒ ∅ | re1 ⇒ ∅ | re_sy _ _ ⇒ {re1}
 | re_union x y ⇒ (π x) ∪ (π y)
 | re_conc x y  ⇒ ((π x)⊙ y) ∪ (π y)
 | re_star x ⇒ ⊙ (π x) (re_star x)
end.
```

The proof that $\pi(\alpha) \le |\alpha|_\Sigma$ is provided by Theorem PI_upper_bound, where the function $|\_|_\Sigma$ is defined by structural induction.

```
Fixpoint |α:re|_Σ : ℕ :=
 match r with
 | re0 ⇒ 0 | re1 ⇒ 0 | re_sy s _ ⇒ 1
 | re_union x y ⇒ (|x|_Σ) + (|y|_Σ)
 | re_conc x y ⇒ (|x|_Σ) + (|y|_Σ)
 | re_star x ⇒ |x|_Σ
 end.

Lemma PI_upper_bound : ∀ r:re, cardinal(π r) ≤ |r|_Σ.
```

A support of a re is a set of re's whose languages verify the right side of equations (2). It is defined by the type MSupport which has two constructors.

```
Inductive MSupport (r:re)(s:re_set) : language :=
| mb_empty : ∀ w:word,  w ∈ ε(r) → w ∈(MSupport r s)
| mb :  ∀ (w:word) (a:sy) (H:IsSy a),  ¬Empty s →
   {x:re & {s':re_set | x ∈ s ∧ w ∈ ((re_sy a H) · x) ∧
       ((re_sy a H) · x) ⊆ r ∧ s' ⊆ s ∧ x =_ℒ LS s')}} →
         w ∈ (MSupport r s).
```

The first constructor, mb_empty, corresponds to the case where a word w belongs to $\varepsilon(r)$. The second constructor, mb, corresponds to the case where a word w belongs to one of the other parcels of the summation in the right side of equations (2). It has a Σ-type as argument which is a witness, $(x,(s',P))$, that $P\,x\,s'$ is a proof that for each $a \in \Sigma$, the parcel $a \cdot \alpha_{il}$ is such that α_{il} is built from an $s' \subseteq s$. The argument $\neg$Empty s is introduced for technical reasons only to facilitate proof construction.

The last, and main result, is the proof that π calculates the support for a given re. This is established in the following theorem:

```
Theorem PI_is_MSupport : ∀ r:re, r =_ℒ MSupport r (π r).
```

The proof of this theorem follows the original proof provided by Mirkin. The proof is constructed by induction on α.

6 Related Work and Applications

Formalization of finite automata in Coq was first approached by J.-C. Filliâtre in [Fil97]. The author's aim was to prove the *pumping lemma* for regular languages and the extraction of an OCaml program. More recently, S. Briais [Bri08] developed a new formalization of formal languages in Coq, which covers Filliâtre's work. This formalization includes Thompson construction of an automaton from a re and a naïve construction of two automata equivalence based in testing if the difference of their languages is the empty language. Braibant and Pous [BP09] formalized KA based on Kozen's algebraic proof of completeness of KA, and provided reflexive tactics to automatically decide KA expression's equivalence. Pereira and Moreira [PM08] also presented a formalization of KA and KAT in Coq. Kleene algebra with tests (KAT) [Koz97] extends KA with an embedded *Boolean algebra* and it is particularly suited for the formal verification of propositional programs. In particular, KAT subsumes *propositional Hoare logic* (PHL) [KT00], a Hoare logic without the assignment axiom. Pereira and Moreira provided a mechanically verified proof that the deductive rules of PHL are theorems of KAT. However, no automation mechanisms were considered.

7 Concluding Remarks

In this paper we have described a formalization of regular languages in the Coq proof assistant. Our main result is the correctness of Mirkin's construction of partial derivative automaton from a regular expression. The overall formalization consists of approximately 2700 lines of specification code, and approximately 7900 lines of proof code.

The results of this paper provide the base for the correctness of a decision procedure for re equivalence, based on the notion of derivative. This kind of procedure was presented by several authors [Brz64,AM95,AMR09a,AMR09b].

This work is the continuation of previous work on the formalization of KA, KAT, and PHL in Coq. Since Coq is a proof assistant that allows the compilation of proofs into binary proof objects, we envision the representation of propositional programs and their properties in the context of Proof-Carrying-Code [Nec97]. In this context programs are packaged together with the certificates that assert program partial correctness.

7.1 Acknowledgments

We thank Sebastien Briais for his willingness to respond promptly to questions concerning his implementation of formal languages in the Coq proof assistant. This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI and the project RESCUE (PTDC/EIA/65862/2006). David Pereira is funded by FCT grant SFRH/BD/33233/2007.

References

[AM95] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theor. Comput. Sci.*, 143(1):51–72, 1995.

[AMR09a] M. Almeida, N. Moreira, and R. Reis. Antimirov and Mosses's rewrite system revisited. *International Journal Of Foundations Of Computer Science*, 20(04):669 – 684, 2009.

[AMR09b] M. Almeida, N. Moreira, and R. Reis. Testing equivalence of regular languages. In *DCFS'09*, Magdeburg, Germany, 2009.

[Ant96] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155(2):291–319, 1996.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BMMR] S. Broda, A. Machiavelo, N. Moreira, and R. Reis. On the average number of states of partial derivative automata. Submitted.

[BP09] T. Braibant and D. Pous. A tactic for deciding Kleene algebras. First Coq Workshop, August 2009. (Available as a HAL report).

[Bri08] S. Briais. Finite automata theory in Coq. Website, 2008. `http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz`.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, October 1964.

[CZ01] J. M. Champarnaud and D. Ziadi. From Mirkin's prebases to Antimirov's word partial derivatives. *Fundam. Inform.*, 45(3):195–205, 2001.

[Fil97] J.-C. Filliâtre. Finite automata theory in Coq - a constructive proof of Kleene's theorem, 1997.

[How80] W. A. Howard. *The formulae-as-types notion of construction*, pages 479–490. Academic Press, 1980.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[Koz94] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[Koz97] D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[KT00] D. Kozen and J. Tiuryn. On the completeness of propositional Hoare logic. In *RelMiCS*, pages 195–202, 2000.

[Mir66] B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:51—57, 1966.

[MPdS09] N. Moreira, D. Pereira, and S. M. de Sousa. On the mechanization of Kleene algebra in Coq. Technical Report DCC-2009-03, DCC-FC&LIACC, Universidade do Porto, 2009.

[Nec97] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[PM93] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, 664:328–345, 1993.

[PM08] D. Pereira and N. Moreira. KAT and PHL in Coq. *Computer Science and Information Systems*, 05(02), December 2008. ISSN: 1820-0214.

[SU98] M. Srensen and P. Urzyczyn. Lectures on the Curry-Howard isomorphism, 1998.