On the mechanization of Kleene Algebra in Coq

Nelma Moreira David Pereira Simão Melo de Sousa

Technical Report Series: DCC-2009-03
Version 1.0 April 2009

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
`http://www.dcc.fc.up.pt/Pubs/`

On the mechanization of Kleene Algebra in Coq

Nelma Moreira David Pereira * Simão Melo de Sousa
{nam,dpereira}@ncc.up.pt,desousa@di.ubi.pt

Abstract

Kleene algebra (KA) is an algebraic system that captures properties of several important structures arising in Computer Science like automata and formal languages, among others. In this paper we present a formalization of regular languages as a KA in the Coq theorem prover. In particular, we describe the implementation of an algorithm for deciding regular expressions equivalence based on the notion of *derivative*. We envision the usage of (an extension of) our formalization as the formal system in which we can encode and prove *proof obligations* for the mechanization and automation of the process of formal software verification, in the context of the Proof Carrying Code paradigm.

1 Introduction

Kleene algebra, (KA) normally called *the algebra of regular events*, is an algebraic system that axiomatically captures properties of several important structures arising in Computer Science, and has been applied in several contexts like automata and formal languages, semantics and logic of programs, design and analysis of algorithms, among others. *Kleene algebra with tests* (KAT) [Koz97b] extends KA with an embedded *Boolean algebra* and is particularly suited for the formal verification of propositional programs. In particular, KAT subsumes *propositional Hoare logic* [KT00], a weaker kind of Hoare logic without the assignment axiom. The formalization of KA, KAT, and of propositional Hoare logic for the Coq theorem prover was presented by us in Pereira and Moreira [PM08].

In this paper we present a formalization of formal languages in the Coq theorem prover. Our contribution is twofold : first, we proved that the set of regular languages is a KA; second, we describe an ongoing work on the implementation of Antimirov and Mosses' algorithm [AM95] for deciding the equivalence of regular expressions, based in the notion of *derivative* of a regular expression. This leads to a decidable procedure for the equational theory of KA.

Our motivation for this work comes from the fact that we envision the usage of (an extension of) our formalization as the formal system where we can be encode and prove *proof obligations* in the context of *Design by Contract* [Mey92]. Considering Kozen's recent work on the decidability of KAT [Koz08], and in the mechanization and automation of program verification for Proof Carrying Code [Nec97].

This paper is organized as follows: in Section 2 we recall some basic definitions of regular languages and KA; in Section 3 we give a brief overview of the Coq theorem prover; in Section 4 we briefly review our previous formalization of KA in Coq; in Sections 5 and 6 we describe the formalization of formal languages and regular expression in Coq, and their integration

*This work was partially funded by Funda cão para a Ciência e Tecnologia (FCT) and program POSI, and by RESCUE project PTDC/EIA/65862/2006.

in our previous formalization of KA; in Section 7 we describe the ongoing formalization of Antimirov and Mosses' decision procedure; finally, in Section 8 we draw our main conclusions, discuss some applications of this work, and point to some current and future work.

2 Preliminaries

We now recall some basic definitions of formal languages and KA that we need throughout the paper. For further details we point the reader to the works of Hopcroft *et al.* [HMU00] and Kozen [Koz97a].

An alphabet Σ is a nonempty set of symbols. A word w over an alphabet Σ is a finite sequence of symbols of Σ. The empty word is denoted by ϵ and the length of a word w is denoted by $|w|$. The concatenation $\cdot$ of two words w_1 and w_2 is a word $w = w_1 \cdot w_2$ obtained by juxtapose the symbols of w_2 after the last symbol of w_1. The set Σ^* is the set of all words over Σ. The triple $(\Sigma^*, \cdot, \epsilon)$ is a monoid.

A language L is subset of Σ^*. 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. For $n \geq 0$, the n^{th} power of a language L is inductively defined by $L^0 = \{\epsilon\}$, $L^n = LL^{n-1}$. The Kleene's *star* L^* of a language L, is $\cup_{n\geq 0} L^n$. A regular expression (r.e.) r over Σ represents a regular language $L(r) \subseteq \Sigma^*$ and is inductively defined by: $\emptyset$ is a r.e and $L(\emptyset) = \emptyset$; ϵ is a r.e and $L(\epsilon) = \{\epsilon\}$; $a \in \Sigma$ is a r.e and $L(a) = \{a\}$; if r_1 and r_2 are r.e., $(r_1 + r_2)$, $(r_1 r_2)$ and $(r_1)^*$ are r.e., respectively with $L((r_1 + r_2)) = L(r_1) \cup L(r_2)$, $L((r_1 r_2)) = L(r_1)L(r_2)$ and $L((r_1)^*) = L(r_1)^*$. We adopt the usual convention that * has precedence over $\cdot$, and $\cdot$ has higher priority than $+$, and we omit outer parentheses. Let *RegExp* be the set of regular expressions over Σ, and let Reg_Σ be the set of regular languages over Σ. Two regular expressions r_1 and r_2 are equivalent if $L(r_1) = L(r_2)$, and we write (the equation) $r_1 = r_2$. The equational properties of regular expressions are axiomatically captured by a KA, normally called *the algebra of regular events*, after the seminal work of S.C. Kleene [Kle].

A KA is an algebraic structure $\mathcal{K} = (K, 0, 1, +, \cdot, ^*)$ such that $(K, 0, 1, +, \cdot)$ is an *idempotent semiring* and where the operator * (Kleene's *star*) is characterized by a set of axioms. We also assume a relation $\leq$ on K, defined by $a \leq b \Leftrightarrow_{def} a + b = b$, for any $a, b \in K$.

There are several ways of axiomatizing a KA. Here we follow the work presented by Dexter Kozen in [Koz94]. The axiomatization we are going to consider has the advantage of being sound over non-standard interpretations, and leads to a complete deductive system for the universal Horn theory of KA (the set of universally quantified equational implications of the form $\wedge_{i=1}^n \alpha_i = \beta_i \to \alpha = \beta$).

In particular, it leads to a decidable procedure for reasoning equationally in KA, as the equational theories of several classes of KA are the same and equal to the one of r.e.'s, *i.e.*, r.e.'s form a KA under the homomorphic *canonical interpretation* $R_\Sigma : RegExpr \to Reg_\Sigma$, such that $R_\Sigma(a) = \{a\}$, for all symbols $a \in \Sigma$.

The set of axioms considered in Kozen's axiomatization are the axioms that characterize idempotent semiring, plus the following that characterize the behavior of Kleene's star:

$$\begin{array}{cc} 1 + xx^* \leq x^* & 1 + x^*x \leq x^* \\ z + yx \leq x \to y^*z \leq x & z + yx \leq x \to y^*z \leq x \end{array} \tag{1}$$

for all $x, y, z \in K$.

3 The Coq interactive theorem prover

The Coq interactive theorem prover [BC04] is an implementation of the *Calculus of Inductive Constructions* (CIC for short) [PM93], a typed λ-calculus with a primitive notion of *inductive types*. An inductive type is a collection of *constructors*, each with its own arity. Each inductive definition also comes with an elimination principle.

Coq's purpose is to allow the mechanization of the process of mathematical theories formalization. Typically, the mathematical objects under study and their basic properties (e.g. axioms) are first specified, and afterwards logical properties that characterize the theories are defined and proved. Coq provides a language and tools for all these formalization steps. Due to its underlying theory (dependent types, higher order functions, and the *Curry Howard isomorphism*), all these tasks can be unified, and the language Gallina allows both the construction of specifications and proofs, in an uniform way.

In the *Curry-Howard isomorphism* principle [SU98, How], any typing relation $t : A$ can either be seen as stating that t has type A, or as stating that t is a proof of the proposition A. Any type in Coq is of one of three kinds of *sorts* : *Set*, *Prop* and *Type*. The first two correspond to the informational and logical terms, respectively. Both belong to the *Type* sort.

Coq supports the definition of complex data structure (e.g. dependent and inductive types) and provably-terminating higher order functions where recursion is obtained by a fixpoint operator guarded by a structurally decreasing argument. Coq also allows users to express higher order properties, and build their proofs. The proofs are terms of Gallina and have a binary representation when compiled, denoted by *proof objects*.

The basic way of the Coq proof construction process is to explicitly build the CIC term corresponding to the proof we are interested in. However, proof can be built more conveniently and interactively in a backward fashion. This step by step process is done by the use of *tactics*. COQ provides a rich tactical language Ltac that allows the construction of proof strategies upon *tactics*.

Most of our formalization uses Coq *module system*. This allows to define both *module types*, and the usual notion of *modules*. A module type is a *signature* of a theory, that specifies the parameters and axioms that describe that theory. In this context axioms refer to properties that must be true in any implementation of that theory. Modules are collections of components that form an implementation. Modules can be parametrized by other modules and, in this case, act as *functors*.

The formalization of the decision procedure we describe in this paper makes use of the *proof by reflection* technique. The idea of this technique in Coq, is to translate Gallina propositions into terms of inductive types representing syntax, so that functions (with the corresponding proofs of correctness) can analyse them. These functions replace the usual deduction steps by computations, which results in smaller proof terms.

4 KA in Coq

In Pereira and Moreira [PM08], we have presented a formalization of KA in the Coq theorem prover. We provided a module signature defining a KA, whose module type is the following:

```
Module Type KA_sig.

Parameter K: Set.
Parameter K0 K1: K.
```

```
Parameter Kstar: K → K.
Parameter Kplus Kdot: K → K → K.

Definition Kleq(x y:K):= Kplus x y = y.

Parameter Is_idemp_semi_ring: idemp_semi_ring_theory K0 K1 Kplus Kdot.

Parameter Is_ka: ka_theory K1 Kstar Kplus Kdot Kleq.

End KA_sig.
```

A module M satisfying the KA_sig signature must implement a type K on which the KA operators are defined, and proofs that Id_idemp_semi_ring and Is_ka are theorems must be provided. Each of these parameters is expected to be a Coq *record*, that is, inductive predicate that has one contructor that takes as arguments the operators defined on K and also all the proofs necessary to verify the axiomatization of KA we have considered. Here we present the definition of the ka_theory record, where the operators $+$, $\cdot$ and * of KA are denoted by [+], [.] and [*], respectively.

```
Record ka_theory:Prop := mk_ka {
 star_ax_1: ∀ x, 1 [+] x[.](x[*]) = x[*];
 star_ax_2: ∀ x, 1 [+] (x[*])[.]x = x[*];
 star_ax_3: ∀ x y z, ((z [+] y[.] x) ≤ x) → (y[*][.]z ≤ x);
 star_ax_4: ∀ x y z, ((z [+] x[.]y) ≤ x) → (z[.]y[*] ≤ x)
}.
```

Our formalization of KA also includes a module with theorems of some properties which are commonly used to reason about KA equalities.

5 Formalization of formal languages

In this section we describe our encoding of formal languages in Coq. To build this theory we have used the Coq modules Lists and Ensembles of the standard library. In the Ensembles module, a set of elements of type X is encoded as the characteristic predicate Ensemble X := X → Prop.

Filliâtre in [Fil97] has developed a formalization of formal languages in Coq that included a constructive proof of Kleene's theorem for regular languages. Our formalization of formal languages is partially based on that work. However, Filliâtre's implementation included an encoding of finite sets that now can be replaced by standard library modules. Our goal in this paper is to consider regular languages as models of KA and integrating it with the work presented in the previous section.

5.1 Alphabet

An alphabet Σ is defined as a list of symbols of a decidable type symb. We also require that all elements of the type symb are elements of Σ. The module type defining the alphabet is the following:

```
Module Type Alphabet.

Parameter symb: Set.
```

```
Parameter symb_eq_dec: ∀ s s',{s=s'}+{s<>s'}.
Parameter Sigma: list symb.
Parameter allSymbolsInSigma: ∀ s,In s Sigma.
Parameter SigmaIsNonEmpty : Sigma <> nil.

End Alphabet.
```

5.2 Words

Words were encoded as lists of elements of type symb. Σ^* is encoded as an inductive predicate on words, representing its characteristic predicate. Word concatenation and the empty word ϵ are represented by list concatenation and by the empty list, respectively. The theorem isValidWord proves that all words built from elements of Σ are elements of Σ^*.

```
Module Words(alph:Alphabet).

Definition word := list symb.

Inductive Sigma_Star: Ensemble word :=
|nil_in_star: In Sigma_Star nil
|add_in_star: ∀ w s,Lists.In s Sigma → In Sigma_Star w →
                        In (Sigma_Star) (s::w).

Theorem isValidWord: ∀ w, In Sigma_Star w.

End Words.
```

5.3 Languages

Languages are terms of the type Ensemble of words. The empty language and the union of languages are the same as the empty set and the union of sets of the type Ensemble. Concatenation of languages, and the Kleene's star of a language were defined along the lines of Filliâtre's work.

```
Module Language(alph:Alphabet).

Module word_props:= Words( alph ).

Definition language:= Ensemble word.

Definition UnionOfLang(x y:language):=  Union x y.

Inductive ConcatOfLang(x y:language): language :=
|conc_sets: ∀ w1 w2,In x w1 → In y w2 → In (ConcatOfLang x y) (w1 ++ w2).

Inductive PowerOfLang(x:language): nat → language :=
|n_0_power: ∀ w,In (Singleton nil) w → In (PowerOfLang x 0) w
|n_n_power: ∀ w1 w2 n,In x w1 → In (PowerOfLang x n) w2 →
                              In (PowerOfLang x (S n)) (w1++w2).

Inductive StarOfLang(x:language): language :=
|star_l: ∀ w n,In (PowerOfLang x n) w → In (StarOfLang x) w.
```

The ConcatOfLang, PowerOfLang and StarOfLang predicates define the concatenation, nth-concatenation and Kleene's star of a language L, respectively. We note that all the operations defined are closed for Σ^*, which is ensured by theorem isValidWord proved in the module Words.

In this module we also prove that the structure $(2^{\Sigma^*}, \cup, \cdot, \{\epsilon\}, \emptyset, ^*)$ is model of KA, by proving the KA axioms. For instance, the following theorems correspond to the to the axioms $0+x = x$ (identity of $+$), $1x = x$ (left identity of $\cdot$), and $(x+y)z = xz+yz$ (right distributivity of $\cdot$ over $+$), respectively.

```
Variables x y z:language.

Theorem lang_union_neutral_left: UnionOfLang Empty_set x = x.

Theorem lang_concat_neutral_left: ConcatOfLang (Singleton nil) x = x.

Theorem lang_concat_distr_left: ConcatOfLang (UnionOfLang x y) z
                     = UnionOfLang (ConcatOfLang x z) (ConcatOfLang x z).
```

To prove the axioms that characterize Kleene's star, we needed several lemmas, including instances of Arden's lemma [DK01].

```
Lemma ka_ax3_aux_1: ∀ n,Included (ConcatOfLang (PowerOfLang y n) z) x →
        Included (ConcatOfLang (StarOfLang y) z) x.

Lemma ka_ax3_aux_2: Included z x /\ Included (ConcatOfLang  y x) x →
        ∀ n, Included (ConcatOfLang (PowerOfLang y n) z) x.

Lemma ka_ax3_aux_3: Included (UnionOfLang (ConcatOfLang y x) z) x →
        Included z x /\  Included (ConcatOfLang y x) x.

Theorem ka_ax_3: Included (UnionOfLang z (ConcatOfLang y x)) x →
          Included  (ConcatOfLang (StarOfLang y) z) x.
```

The theorem ka_ax_3 corresponds to the axiom $z+yx \leq x \rightarrow y^*z \leq x$. With $z+yx \leq x$ as hypothesis, we obtain $z \leq x$ and $yx \leq x$ by ka_ax3_aux_3. By ka_ax3_aux_2 we obtain $\forall n, y^n z \leq x$. Finally, by applying ka_ax3_aux_2 we get $y^*z \leq x$, thus finishing the proof.

We now define a module KaModelLang that satisfies the module type KA_sig, instantiates the abstract operations with the corresponding operations for languages of the Language module, and where the axioms of KA are proved using the theorems just defined. This task is straightforward, and below we present as an example the proof of $x + 0 = 0$. The proof of the rest of the axioms are done in a similar way.

```
Module KaModelLang(alph:Alphabet) : KA_sig.

Module ws := Words(alph).
Module lg := Language(alph).

Definition K := language.
Definition K0 := Empty_set.
Definition K1 := Singleton nil.
Definition Kplus := UnionOfLang.
Definition Kdot := ConcatOfLang.
Definition Kstar := StarOfLang.

Definition Kleq(x y:K):= Kplus x y = y.

Lemma empty_re_left: Kplus K0 x = x.
```

```
Proof.
 intros.
 apply lang_union_neutral_left.
Qed.

Lemma empty_re_right: Kplus x K0 = x.
Lemma absortion_re_left: Kdot K0 x = K0.
Lemma absortion_re_right: Kdot x K0 = K0.
Lemma identity_dot_left: Kdot K1 x = x.
Lemma identity_dot_right: Kdot x K1 = x.
Lemma identity_dot: Kdot x K1 = Kdot K1 x.
Lemma plus_idempotence: Kplus x x = x.
Lemma plus_commutativity: Kplus x y = Kplus y x.
Lemma plus_associativity: Kplus (Kplus x y) z = Kplus x (Kplus y z).
Lemma dot_associativity: Kdot (Kdot x y) z = Kdot x  (Kdot y z).
Lemma dot_distr_right: Kdot x (Kplus  y z) = Kplus (Kdot x y) (Kdot x z).
Lemma dot_distr_left: Kdot (Kplus y z) x = Kplus (Kdot y x) (Kdot z x).

Theorem Is_idemp_semi_ring: idemp_semi_ring_theory K0 K1 Kplus Kdot.

Lemma star_ax_re_1: Kplus K1 (Kdot x (Kstar x)) = Kstar x.
Lemma star_ax_re_2: Kplus K1 (Kdot (Kstar  x) x) = Kstar x.
Lemma star_ax_re_3: Kleq (Kplus z (Kdot y x)) x →
                                          Kleq (Kdot (Kstar y) z) x.
Lemma star_ax_re_4: Kleq (Kplus z (Kdot x y)) x →
                                          Kleq (Kdot z (Kstar y)) x.

Theorem Is_ka: ka_theory K1 Kstar Kplus Kdot Kleq.

End KaModelRegLang.
```

6 Regular expressions as a KA

We have formalized r.e.'s as syntactical representations of regular languages. For that, we have defined an inductive type RegExpr and a fixpoint function from_re_to_lang. The former inductively defines a r.e., while the second builds the regular language corresponding to the r.e. given as input.

```
Module RegExprs(alph:Alphabet).

Inductive RegExpr : Set :=
|empty_re    : RegExpr
|epsilon_re  : RegExpr
|symb_re     : symb →  RegExpr
|plus_re     : RegExpr → RegExpr → RegExpr
|dot_re      : RegExpr → RegExpr → RegExpr
|star_re     : RegExpr → RegExpr.

Fixpoint from_re_to_lang (re:RegExpr) : language :=
match re with
|empty_re         => Empty_set
|epsilon_re       => Singleton nil
|symb_re s        => Singleton (sy::nil)
|plus_re re1 re2 => UnionOfLang   (from_re_to_lang re1) (from_re_to_lang re2)
|dot_re re1 re2  => ConcatOfLang  (from_re_to_lang re1) (from_re_to_lang re2)
|star_re re1      => StarOfLang (from_re_to_lang re1)
end.

(* ... *)
```

The axiom eq_re_rl defines that two regular expressions are equivalent if the language they represent is the same. The function from_re_to_lang is proved correct in theorem all_re_is_regular.

```
Axiom eq_re_rl:∀ r1 r2, from_re_to_lang r1 = from_re_to_lang r2 → r1 = r2.

Inductive RegLang : language → Prop :=
|rl_empty    : RegLang (Empty_set)
|rl_epsilon  : RegLang (Singleton nil)
|rl_symbol   : ∀ s, RegLang (Singleton (s::nil))
|rl_plus     : ∀ l1 l2, RegLang l1 → RegLang l2 → RegLang (UnionOfLang l1 l2)
|rl_dot      : ∀ l1 l2, RegLang l1 → RegLang l2 → RegLang (ConcatOfLang l1 l2)
|rl_star     : ∀ l, RegLang l → RegLang (StarOfLang l).

Theorem all_re_is_regular : ∀ re, RegLang (from_re_to_lang re).

End RegExprs.
```

We have implemented the module KaModelRegExpr that satisfies KA_sig where the parameter K is now defined as being the type RegExpr, the type of r.e.'s. The proof that this module satisfies the signature KA_sig follows the same steps we have taken for the module KaModelLang.

```
Module KaModelRegExpr( alph : Alphabet) : KA_sig.

Module ws := Words(alph).
Module re := RegExprs(alph).

Definition K     := RegExpr.
Definition K0    := empty_re.
Definition K1    := epsilon_re.
Definition Kplus := plus_re.
Definition Kdot  := dot_re.
Definition Kstar := star_re.

Definition Kleq( x y : K) := Kplus x y = y.

Lemma empty_re_left : Kplus 0 x = x.
Proof.
 intros.
 apply eq_re_rl.
 apply lang_union_neutral_left.
Qed.

(* ... *)

End KaModelRegExpr.
```

7 The decision procedure

The usual procedure for determining that two regular expressions are equivalent is to transform each regular expression in an equivalent minimal finite automaton, and decide if the resulting automata are isomorphic [HMU00]. Kozen completeness theorem of the axiomatization presented in Section 2 for the algebra of regular events is based on considering finite automata over KA and using that usual procedure.

Antimirov and Mosses [AM95] proposed a complete and terminating rewrite system for deciding the equivalence of two r.e.'s. This rewrite system is based on the notion of derivatives of regular expressions. Testing the equivalence of two r.e.'s corresponds to an iterated process of testing the equivalence of their derivatives. Termination is ensured because the set of derivatives to be considered is finite, and possible cycles are detected using *memoization*. Almeida *et al.* in [MAR08] presented an improved functional version of the Antimirov and Mosses (AM) method. In the next subsection we review some notions of derivatives of r.e.'s and present the AM method. In the subsequent subsections we present an ongoing work on the formalization of that method in the Coq theorem prover.

7.1 Derivatives and equivalence of regular expressions

The *derivative* [Brz64] of a r.e. α with respect to a symbol $a \in \Sigma$, denoted by $a^{-1}(\alpha)$, is inductively defined on the structure of α as follows:

$$\begin{array}{rclrcl} a^{-1}(\emptyset) & = & \emptyset, & a^{-1}(\alpha+\beta) & = & a^{-1}(\alpha)+a^{-1}(\beta) \\ a^{-1}(\epsilon) & = & \emptyset, & a^{-1}(\alpha\beta) & = & a^{-1}(\alpha)\beta+\varepsilon(\alpha)a^{-1}(\beta) \\ a^{-1}(b) & = & \begin{cases} \epsilon, & \text{if } b=a; \\ \emptyset, & \text{otherwise} \end{cases} & a^{-1}(\alpha^*) & = & a^{-1}(\alpha)\alpha^* \end{array}$$

where $\varepsilon(\alpha)$ is called the *constant part* of α and is defined as follows: $\varepsilon(\alpha)=\epsilon$ if $\epsilon \in L(\alpha)$, and $\varepsilon(\alpha)=\emptyset$ otherwise.

In particular, we have that for any r.e. α,

$$\alpha = \varepsilon(\alpha) + \sum_{a\in\Sigma} a^{-1}(\alpha) \tag{2}$$

This notion of derivative can be easily extended to words $w \in \Sigma^*$, denoted $w^{-1}(\alpha)$, which is inductively defined on the structure of w as follows:

$$\begin{array}{rcl} \epsilon^{-1}(\alpha) & = & \alpha \\ (ua)^{-1}(\alpha) & = & a^{-1}(u^{-1}(\alpha)), \text{for any } u \in \Sigma^* \end{array} \tag{3}$$

Considering r.e. modulo the ACI axioms (associativity (A), commutativity (C) and idempotence (I) of $+$), Brzozowski [Brz64] proved that the set of derivatives of a r.e. α is finite. We will now present a version of Antimirov and Mosses' method that is basically the approach proposed by Almeida *et al.* in [MAR08], and that we denote by the algorithm AM. The algorithm takes as input a pair of r.e.'s (α,β) and returns *True* if and only if $\alpha=\beta$.

$S = \{\alpha, \beta\}$
$H = \emptyset$
while $(\alpha,\beta) = POP(S)$ **do**
  if $\varepsilon(\alpha) \neq \varepsilon(\beta)$ **then**
    return False
  end if
  $PUSH(H,(\alpha,\beta))$
  for $a \in \Sigma$ **do**
    $\alpha' = a^{-1}(\alpha)$
    $\beta' = a^{-1}(\beta)$
    if $(\alpha',\beta') \notin H$ **then**
      $PUSH(S,(\alpha',\beta'))$

        end if
    end for
end while
return True

The set S collects the pairs of derivatives to be tested, and the set H is used to prevent the algorithm to loop, by testing r.e.'s equality modulo ACI. Considering the equivalence 2, the algorithm successively tests the pairs of derivatives. When either a pair of derivatives is such that their constant parts are different, or the set S is empty the algorithm terminates.

The following theorems and lemmas ensure the correctness of the method [MAR08].

Theorem 1. *The algorithm AM is terminating.*

Lemma 1. *If* $\alpha = \beta$ *then* $a^{-1}(\alpha) = a^{-1}(\beta)$*, for all* $a \in \Sigma$*.*

Theorem 2. *The algorithm returns True if and only if* $\alpha = \beta$*.*

We note that this decision procedure corresponds to the co-algebraic approach based on deterministic automata of Rutten [Rut98], and that were extended to KAT by Chen and Purcella [CP04], and by Kozen [Koz08]. Moreover, the relation between this method and the one based on automata was recently approached by Almeida *et al.* [AMR09]. In particular, the complexity of this decision procedure can be made at least as efficient as the construction of non-deterministic finite automata from a r.e., obtaining the equivalent deterministic finite automata, and determining a bisimulation between them [Koz94].

7.2 Regular expressions modulo ACI

We have implemented normalization of r.e.'s modulo ACI along the lines of the formalization presented in [BC04] (Chapter 16), for normalizing numerical expressions modulo AC for addition. The underlying idea is to obtain a normal form for r.e. such that two r.e.'s are equal modulo ACI if and only if their normal form is the same.

Consider the syntactic tree associated with a r.e.. For associativity the usual normalization procedure is to consider a flattened binary tree where the nodes representing the $+$ operator have as left child either a leaf, or a sub-tree whose root does not represent the operator $+$. To deal with the commutativity we must provide an order relation on the nodes representing the operators of r.e., and sort the trees according to this order. Idempotence is achieved by removing the occurrence of repeat elements in the sorted tree. The normalization if proved correct if the r.e. given as input and the normalized r.e.'s are proved equivalent.

Given a RegExpr term, we inductively define a type ptree as follows:

```
Inductive ptree : Set :=
|val_pt : nat → ptree
|pls_pt : ptree → ptree → ptree
|dot_pt : ptree → ptree → ptree
|str_pt : ptree → ptree.
```

The type ptree represents a tree with three kinds of internal nodes, and whose leafs contain natural numbers. The nodes pls_pt, dot_pt and str_pt represent the $+$, $\cdot$ and * operators, respectively. The r.e.'s ϵ, $\emptyset$ and the symbols of the alphabet are represented by val_pt 0, val_pt 1, and val_pt n with $n \geq 2$, respectively. A term of type ptree can be converted back to its corresponding RegExpr term through the function ptree_to_re.

The function flatten_ptree takes a ptree term as argument, and returns a flattened ptree term. The correctness of this function is given by the following facts:

```
Lemma flatten_ptree_valid : ∀ x, ptree_to_re x = ptree_to_re (flatten_ptree x).

Lemma flatten_ptree_valid_2 : ∀ x y,
        ptree_to_re (flatten_ptree x) = ptree_to_re (flatten_ptree y) →
        ptree_to_re x = ptree_to_re y.
```

We have encoded an insertion sort algorithm for sorting an already flattened ptree. The sort_ptree sorting function uses the order val_pt < pls_pt < dot_pt < str_pt on internal nodes, and the $\leq$ order on the natural numbers for sorting between val_pt terms. The correctness of the sorting algorithm is given by the following lemmas:

```
Lemma sort_apt_eq : ∀ t, ptree_to_re (sort_ptree t) = ptree_to_re t.

Theorem sort_apt_eq_re : ∀ t1 t2,
     ptree_to_re (sort_ptree t1) = ptree_to_re (sort_ptree t2) →
     ptree_to_re t1 = ptree_to_re t2.
```

Finally, idempotence is implemented by the tactic idemp_rm that searches for the patterns pls_pt x (pls_pt x _) and pls_pt x x, and replace them by pls_pt x _ and x, respectively.

The process of normalization modulo ACI is done by a special tactic that we have implemented and that given an r.e. equality x = y, converts it to into the equality

$$\begin{gathered}\mathsf{ptree\_to\_re}\ (\mathsf{sort\_ptree}\ (\mathsf{flatten\_ptree}\ \mathsf{t1})) \\ = \\ \mathsf{ptree\_to\_re}\ (\mathsf{sort\_ptree}\ (\mathsf{flatten\_ptree}\ \mathsf{t2}))\end{gathered}$$

and that applies the theorems sort_apt_eq and flatten_ptree_valid_2. Then, by simple computation the original equality modulo AC. Finally, the tactic idemp_rm is executed and the equality between x and y modulo ACI is determined.

7.3 Formalizing derivatives

The derivative of a re is implemented in Coq as the recursive function drv defined on terms ptree. The function epsilon_ptree represents the ε function of AM's algorithm. The derivative of a r.e. is extended to words by the recursive function wdrv.

```
Fixpoint drv (t:ptree) (s:nat) {struct t} : ptree :=
match t with
|val_pt x => match x with
               |O => val_pt O
               |1 => val_pt O
               |n => if eq_nat_dec x s then val_pt 1 else val_pt O
               end
|pls_pt a b => pls_pt (drv a s) (drv b s)
|dot_pt a b => pls_pt (dot_pt (drv a s) b) (dot_pt (epsilon_ptree a) (drv b s))
|str_pt a   => dot_pt (drv a s) (str_pt a)
end.

Fixpoint wdrv (t:ptree) (w:list nat) {struct w} : ptree :=
```

```
match w with
|nil      => r
|(s::ls) => drv (wdrv t ls) s
end.
```

7.4 Formalizing the decision procedure

The recursive function AM_eq implements the algorithm AM presented in Section 7.1. Notice that AM_eq is not defined with the Fixpoint keyword. Instead it is implemented with the Function keyword, plus a parameter measure length x. Like in Fixpoint, the decreasing argument must be given but it must not necessary be structurally decreasing. The role of measure is to name the decreasing argument and to define that the decreasing criteria is the length of x, that is used to ensure termination of recursive calls.

```
Fixpoint drv_of_sigma (tpair:(ptree*ptree)) (x:list nat) (h s:list (ptree*ptree))
                       {struct x}: list (ptree*ptree) :=
match x with
|nil => s
|(a::xs) => let p := (brz_pair a rpair) in
       match In_dec pair_eq p h with
       |left _ =>  brz_of_sigma rpair xs h s
       |right _ => brz_of_sigma rpair xs h (app  s (p::nil))
       end
end.

Function AM_eq (s h:list (ptree*ptree)) {measure length s} : bool :=
match s with
|nil => true
|(x::xs) => match has_epsilon (fst x) (snd  x) with
            |true =>  AM_eq (drv_of_sigma x sigma_ptree) h xs) (x::h)
            |false => false
            end
end.
```

The recursive function drv_of_sigma calculates the set of pairs of derivatives $\{(a^{-1}(\alpha), a^{-1}(\beta)) \mid a \in \Sigma\}$, and adds it to the list containing the pairs of r.e.'s still to be tested. The function has_epsilon returns true if ε(fst x)= ε(snd x), and returns false otherwise. The term sigma_ptree is a list representing the symbols of Σ such that if $a_i \in \Sigma$ then val_pt $(i+2)$ belongs to sigma_tree (recall that val_pt 0 and val_pt 1 represent the r.e.'s $\emptyset$ and ϵ, respectively).

8 Concluding remarks and applications

In this paper we have presented a formalization of regular languages in the Coq theorem prover. The formalization of the correctness of the decision procedure AM will be the subject of a companion paper.

Our research line is to use this framework, and in particular its extension to KAT, as the formal system for expressing and proving *proof obligations* about computer programs. KAT has enough expressivity to represent simple *while*-programs with propositional tests. In particular, KAT subsumes *propositional Hoare logic*. Previous work on the formalization of KAT and propositional Hoare logic has already been done by Pereira and Moreira in [PM08]. There we provided examples of proofs of correctness by manually converting while-programs to KAT terms along the lines of Angus and Kozen's Schematic KAT (SKAT) [AK01].

The usage of a formal system based on KA (and extensions of it) in the context of the *Design by Contract* and *Proof Carrying Code* paradigms is a very appealing subject of research. First because KA and its extensions have very simple and compact representation of their terms and proofs and, second, because they have automatic decision procedures for proving equivalence of terms in their equational theory.

References

[AK01] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.

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

[AMR09] M. Almeida, N. Moreira, and R.Reis. Testing regular languages equivalence, 2009. Submitted.

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

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

[CP04] H Chen and R Pucella. A coalgebraic approach to Kleene algebra with tests. *Theoretical Computer Science*, 327(1-2):23–44, 2004.

[DK01] Ding-Zhu Du and Ker-I Ko. *Problem Solving in Automata, Languages, and Complexity.* Wiley-Interscience, 09 2001.

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

[HMU00] J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 2000.

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

[Kle] Stephen Cole Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J.

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

[Koz97a] Dexter Kozen. *Automata and Computability.* Undergrad. Texts in Computer Science. Springer-Verlag, 1997.

[Koz97b] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[Koz08] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Computing and information science technical reports, Cornell University, March 2008.

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

[MAR08] N. Moreira M. Almeida and R. Reis. Antimirov and Mosses's rewrite system revisited. In O. Ibarra and B. Ravikumar, editors, *CIAA 2008: Thirteenth International Conference on Implementation and Application of Automata*, number 5448 in LNCS, pages 46–56. Springer-Verlag, 2008.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[Nec97] George 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] David Pereira and Nelma Moreira. KAT and PHL in Coq. *Computer Science and Information Systems*, 05(02), December 2008. ISSN: 1820-0214.

[Rut98] Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.

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