Deciding Regular Expressions (In-)Equivalence in Coq

Nelma Moreira, David Pereira, and Simão Melo de Sousa

Technical Report Series: DCC-2011-06
Version 1.0

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

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/

Deciding Regular Expressions (In-)Equivalence in Coq*

Nelma Moreira
DCC-FC & CMUP – University of Porto
Rua do Campo Alegre 1021, 4169-007
Porto, Portugal
nam@dcc.fc.up.pt

David Pereira
DCC-FC & LIACC – University of Porto
Rua do Campo Alegre 1021, 4169-007
Porto, Portugal
dpereira@ncc.up.pt

Simão Melo de Sousa
LIACC & DI – University of Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001
Covilhã, Portugal
desousa@di.ubi.pt

Abstract

In this paper we present a mechanically verified implementation of an algorithm for deciding regular expression (in-)equivalence within the Coq proof assistant. This algorithm is a version of a functional algorithm proposed by Almeida *et al.* which decides regular expression equivalence through an iterated process of testing the equivalence of their partial derivatives. In particular, this algorithm has a refutation step which improves the process of checking if two regular expressions are not equivalent.

1 Introduction

Recently, much attention has been given to the mechanisation of Kleene algebra (KA) within proof assistants. J.-C. Filliâtre [Fil97] provided a first formalisation of Kleene theorem for regular languages [Kle] within the Coq proof assistant [BC04]. Höfner and Struth [HS07] investigated the automated reasoning in variants of Kleene algebras with Prover9 and Mace4 [McC]. Pereira and Moreira [MP08] implemented in Coq an abstract specification of Kleene algebra with tests (KAT) [Koz97] and the proofs that Propositional Hoare logic deduction rules are theorems of KAT. An obvious follow up of that work was to implement a certified procedure for deciding equivalence of KA terms, i.e regular expressions. A first step was the proof of the correctness of the partial derivative automata construction from a regular expression presented in [AMPMdS11]. In this paper, our goal is to mechanically verify a decision procedure based on partial derivatives proposed by Almeida *et al.* [AMR08] that is a functional variant of the rewrite system of Antimirov and Mosses [AM94].

In a different setting, Braibant and Pous [BP10] formally verified Kozen's proof of the completeness of Kleene algebra [Koz94] in Coq. This proof is based on the classic conversion

*This work was partially funded by Funda cão para a Ciência e Tecnologia (FCT) and program POSI, and by the CANTE project PTDC/EIA-CCO/101904/2008. David Pereira is funded by FCT grant SFRH/BD/33233/2007.

of regular expressions into equivalent minimal deterministic finite automata formulated in a algebraic setting.

Independently of the work here presented, Coquand and Siles [CS] mechanically verified an algorithm for deciding regular expression equivalence based on Brzozowski's derivatives [Brz64] using Coq's SSReflect extension [Theb] and an inductive definition of finite sets called Kuratowski-finite sets. Finally, Krauss and Nipkow [KN11] provide an elegant and concise formalisation of Rutten's co-algebraic approach of regular expression equivalence [Rut98], in the Isabelle proof assistant [NPW02], but they do not address the termination of the formalised decision procedure.

Our formalisation differs from the two previous formalisations as it is a refutation method based on partial derivatives. The use of partial derivatives avoids the necessary normalisation of regular expressions modulo *ACI* (i.e associativity, idempotence and commutativity of union) in order to ensure the finiteness of Brzozowski's derivatives. The refutation step improves the detection of inequivalent regular expressions. Similarly to those works, the procedure we have formalised is purely syntactic and does not require the construction of automata. Our long term objective is to use the procedure we have implemented as a way to automate the process of reasoning about programs encoded as KAT terms.

This paper is organised as follows: in Section 2 we recall the basic definitions of regular languages; in Section 3 we show how the decision procedure was formalised and proved correct and complete with respect to regular expression equivalence; finally, in Section 4 we draw our main conclusions and we point to some current and future work.

2 Some basic notions of regular languages

This section presents the basic notions of regular languages that are required to implement our decision procedure. This definitions can be found on standard books such as Hopcroft's *et al.* [HMU00], and their formalisation in the Coq proof assistant are presented by Almeida *et al.* in [AMPMdS11].

2.1 Alphabets, words, languages and regular expressions

Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$ be an *alphabet* (non-empty set of symbols). A *word* w over Σ is any finite sequence of symbols. The *empty word* is denoted by ε and 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 $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. The *power* of a language is inductively defined by $L^0 = \{\varepsilon\}$ and $L^n = LL^{n-1}$, with $n \geq 1$. The *Kleene star* $L^\star$ of a language L is $\cup_{n\geq 0} L^n$. Given a word $w \in \Sigma^\star$, the *(left-)quotient* of L by the word w is the language $w^{-1}(L) = \{v \mid wv \in L\}$.

A *regular expression* (*re*) α over Σ represents a *regular language* $\mathcal{L}(\alpha) \subseteq \Sigma^*$ and is inductively defined by: $\emptyset$ is a *re* and $\mathcal{L}(\emptyset) = \emptyset$; ε is a *re* and $\mathcal{L}(\varepsilon) = \{\epsilon\}$; $\forall\, a \in \Sigma$, a is a *re* and $\mathcal{L}(a) = \{a\}$; if α and β are *re*'s, $(\alpha + \beta)$, $(\alpha\beta)$ and $(\alpha)^*$ are *re*'s, respectively with $\mathcal{L}(\alpha + \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$ and $\mathcal{L}(\alpha^\star) = \mathcal{L}(\alpha)^\star$. If Γ is a set of *re*'s, then $\mathcal{L}(\Gamma) = \cup_{\alpha\in\Gamma}\mathcal{L}(\alpha)$. The *alphabetic size* of a *re* α is the number of symbols of the alphabet in α and is denoted by $|\alpha|_\Sigma$. The *empty word property* (*ewp* for short) of a *re* α is denoted by $\varepsilon(\alpha)$ and is defined by $\varepsilon(\alpha) = \varepsilon$ if $\varepsilon \in \mathcal{L}(\alpha)$ and by $\varepsilon(\alpha) = \emptyset$, otherwise. If $\varepsilon(\alpha) = \varepsilon(\beta)$ we say that α and β *have the same ewp*. Given a set of *re*'s Γ we define $\varepsilon(\Gamma) = \varepsilon$ if there exists a *re* $\alpha \in \Gamma$ such that $\varepsilon(\alpha) = \varepsilon$ and $\varepsilon(\Gamma) = \emptyset$, otherwise. Two *re*'s α and β are *equivalent* if they represent the same language, that is, if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, and we write $\alpha \sim \beta$.

2.2 Partial derivatives

The notion of *derivative* of a *re* was introduced by Brzozowski [Brz64]. Antimirov [AM94] extended this notion to the one of set of *partial derivatives*, which correspond to a finite set representation of Brzozowski's derivatives.

Let α be a *re* and let $a \in \Sigma$. The set $\partial_a(\alpha)$ of partial derivatives of the *re* w.r.t. the symbol a is inductively defined as follows:

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

where $\Gamma\beta = \{\alpha\beta \mid \alpha\in\Gamma\}$ if $\beta \neq \emptyset$ and $\beta\neq\varepsilon$, and $\Gamma\emptyset=\emptyset$ and $\Gamma\varepsilon=\Gamma$ otherwise (in the same way we define $\beta\Gamma$). Moreover one has

$$\mathcal{L}(\partial_a(\alpha)) = a^{-1}(\mathcal{L}(\alpha)) \tag{1}$$

The definition of partial derivative is extended to sets of *re*'s and to words. Given a *re* α, a symbol $a\in\Sigma$, a word $w\in\Sigma^\star$, and a set of *re*'s Γ, we define $\partial_a(\Gamma)=\cup_{\alpha\in\Gamma}\partial_a(\alpha)$, $\partial_\varepsilon(\alpha)=\{\alpha\}$, and $\partial_{wa}=\partial_a(\partial_w(\alpha))$. Equation (1) can be extended to words $w\in\Sigma^\star$. The *set of partial derivatives* of a *re* α is defined by $PD(\alpha)=\cup_{w\in\Sigma^\star}(\partial_w(\alpha))$. This set is always finite and its cardinality is bounded by $|\alpha|_\Sigma+1$.

Champarnaud and Ziadi show in [CZ01] that partial derivatives and Mirkin's prebases [Mir66] lead to identical constructions. Let $\pi(\alpha)$ be a function inductively defined as follows:

$$\begin{array}{lcllcl}
\pi(\emptyset) & = & \emptyset & \pi(\alpha+\beta) & = & \pi(\alpha)\cup\pi(\beta) \\
\pi(\varepsilon) & = & \emptyset & \pi(\alpha\beta) & = & \pi(\alpha)\beta\cup\pi(\beta) \\
\pi(a) & = & \{\varepsilon\} & \pi(\alpha^\star) & = & \pi(\alpha)\alpha^\star
\end{array} \tag{2}$$

In his original paper, Mirkin proved that $\#\pi(\alpha)\leq|\alpha|_\Sigma$, while Champarnaud and Ziadi established that $PD(\alpha)=\{\alpha\}\cup\pi(\alpha)$. These properties were proven correct in Coq by Almeida *et al.* [AMPMdS11] and will be used to prove the termination of the decision procedure described in this paper.

An important property of partial derivatives is that given a *re* α we have

$$\alpha\sim\varepsilon(\alpha)+\sum_{a\in\Sigma}a\partial_a(\alpha) \tag{3}$$

and so, checking if $\alpha\sim\beta$ can be reformulated as

$$\varepsilon(\alpha)+\sum_{a\in\Sigma}a\partial_a(\alpha)\sim\varepsilon(\beta)+\sum_{a\in\Sigma}a\partial_a(\beta). \tag{4}$$

This will be an essential ingredient to our decision method because deciding if $\alpha\sim\beta$ is tantamount to check if $\varepsilon(\alpha)=\varepsilon(\beta)$ and if $\partial_a(\alpha)\sim\partial_a(\beta)$, for each $a\in\Sigma$. We also note that testing if a word $w\in\Sigma^\star$ belongs to $\mathcal{L}(\alpha)$ can be reduced to the purely syntactical operation of checking if

$$\varepsilon(\partial_w(\alpha))=\varepsilon. \tag{5}$$

By (4) and (5) we have that

$$(\forall w\in\Sigma^\star,\varepsilon(\partial_w(\alpha))=\varepsilon(\partial_w(\beta)))\leftrightarrow\alpha\sim\beta \tag{6}$$

3 The decision procedure

In this section we describe the implementation in Coq of a procedure for deciding the equivalence of *re*'s based on partial derivatives. First we give the informal description of the procedure and afterwards we present the technical details of its implementation in Coq's type theory. The Coq development of the decision procedure presented in this paper is available online in [MPaM].

3.1 Informal description

The procedure for deciding the equivalence of *re*'s, which we call `equivP`, is presented in Fig.1. Given two *re*'s α and β this procedure corresponds to the iterated process of deciding the equivalence of α and β by computing the equivalence of their derivatives, in the way noted in equation (4). The function `equivP` works over pairs of *re*'s (Γ, Δ) such that $\Gamma = \partial_w(\alpha)$ and $\Delta = \partial_w(\beta)$, for some word $w \in \Sigma^\star$. From now on, we refer to these pairs simply by derivatives. To check if $\alpha \sim \beta$ it is enough to test the *ewp* 's of the derivatives, *ie.*, if (Γ, Δ) verify the condition

$$\varepsilon(\Gamma) = \varepsilon(\Delta) \tag{7}$$

```
S={({α},{β})}
H=∅
def equivP(H,S):
 while S ≠ ∅ :
   (Γ,Δ) = POP(S)
   if ε(Γ) ≠ ε(Δ):
     return false
   else:
    H = H ∪ {(Γ,Δ)}
    for a ∈ Σ:
     (Λ,Θ) = ∂_a(Λ,Δ)
     if (Λ,Θ) ∉ H :
      S = S ∪ {(Λ,Θ)}
 return true
```

Figure 1: The procedure `equivP`.

Two finite sets of derivatives are required for implementing `equivP`: a set H that serves as an accumulator for the derivatives already processed by the procedure, and a set S which serves as a working set that gathers new derivatives yet to be processed. The set H ensures the termination of `equivP` due to the finiteness of the number of derivatives.

When `equivP` terminates, either the set H of all the derivatives of α and β has been computed, or a counter-example (Γ, Δ) has been found, *ie.*, $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$. By equation (6), in the first case we conclude that $\alpha \sim \beta$ and, in the second case we conclude that $\alpha \not\sim \beta$. The correctness of this method can be found in Almeida *et al.* [AMR08, AMR10].

3.2 Implementation in Coq

In this section we describe the mechanically verified formalisation of `equivP` in the Coq proof assistant and show its termination and correctness.

3.2.1 Certified pairs of derivatives.

The main data structures underlying the implementation of `equivP` are pairs of sets of *re*'s and sets of these pairs. Each pair (Γ, Δ) corresponds to a word derivative $(\partial_w(\alpha), \partial_w(\beta))$, where $w \in \Sigma^\star$ and α and β are the *re*'s being tested by `equivP`. The pairs (Γ, Δ) are encoded by the type `ReW` α β, presented in Fig.2. This is a *dependent record* built from three parameters: a pair of sets of *re*'s `dp` that corresponds to the actual pair (Γ, Δ), a word `w`, and a proof term `cw` that certifies that $(\Gamma, \Delta) = (\partial_{\mathtt{w}}(\alpha), \partial_{\mathtt{w}}(\beta))$. The dependency of `ReW` α β comes from `cw`, which is a proof depending on the values of the *re*'s α and β, and on the word parameter `w`. This dependency ensures, at compilation time, that `equivP` will only accept as input pairs of *re*'s that correspond to derivatives of α and β.

```
Record ReW (α β:re) := mkReW {
 dp :> set re * set re ;
 w  : word ;
 cw : dp === (∂w(α),∂w(β))
}.

Program Definition ReW_1st (α β:re) : ReW α β.
refine(Build_ReW ({r1},{r2}) nil _).
(* Proof that ({α},{β}) = (∂ε(α),∂ε(β)) *).
Defined.

Definition ReW_pdrv(α β:re)(x:ReW α β)(a:A) : ReW α β.
refine(match x with
       | mkReW α β K w P => mkReW α β (pdrvp K a) (w++[a]) _
       end).
(* Proof that ∂a(∂w(α),∂w(β)) = (∂wa(α),∂wa(β)) *)
Defined.

Definition ReW_pdrv_set(s:ReW α β)(sig:set A) : set (ReW α β) :=
 fold (fun x:A => add (ReW_pdrv s x)) sig ∅.

Definition ReW_wpdrv (α β:re)(w:word) : ReW α β.
refine(mkReW α β (∂w(α),∂w(β)) w _).
reflexivity.
Defined.

Definition c_of_rep(x:set re * set re) :=
 Bool.eqb (c_of_re_set (fst x)) (c_of_re_set (snd x)).

Definition c_of_ReW(x:ReW α β) := c_of_rep (dp x).

Definition c_of_ReW_set (s:set (ReW α β)) : bool :=
 fold (fun x => andb (c_of_ReW x)) s true.
```

Figure 2: Definition of the type `ReW` and the extension of derivatives and *ewp* functions.

The type `ReW` α β provides also an easy way to relate the computation of `equivP` and the equivalence of α and β: if H is the set returned by `equivP`, then the equation (6) is tantamount to check the *ewp* of the elements of H. Furthermore, using this type provides a simple way of keeping the set of words from which the set of derivatives of α and β has been obtained. For that it is enough to apply the projection `w` to each pair $(\Gamma, \Delta) \in H$.

The notions of derivative and of *ewp* are extended to the type `ReW` α β as implemented by the functions `ReW_pdrv` and `c_of_ReW`, and to sets of terms `ReW` α β by the functions `ReW_pdrv_set` and `c_of_ReW_set`, respectively.

3.2.2 Computation of new derivatives.

The *while-loop* of `equivP` describes the process of testing the equivalence of the derivatives of α and β. In each iteration of this process, new derivatives (Γ, Δ) are computed until either the working set S becomes empty, or a pair (Γ, Δ) such that $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$ is found. This is precisely what the function `step` presented in Fig.3 does.

```
Definition ReW_pdrv_set_filtered(x:ReW α β)(H:set (ReW α β))
 (sig:set A) : set (ReW α β) :=
  filter (fun y => negb (y ∈ H)) (ReW_pdrv_set x sig).

Inductive step_case (α β:re) : Type :=
|proceed   : step_case α β
|termtrue  : set (ReW α β) → step_case α β
|termfalse : ReW α β → step_case α β.

Definition step (H S:set (ReW α β))(sig:set A) :
 ((set (ReW αβ) * set (ReW α β)) * step_case α β) :=
 match choose s with
 |None => ((H,S),termtrue α β H)
 |Some (Γ,Δ) =>
   if c_of_ReW _ _ (Γ,Δ) then
    let H' := add (Γ,Δ) H in
     let S' := remove (Γ,Δ) S in
      let ns :=  ReW_pdrv_set_filtered α β (Γ,Δ) H' sig in
       ((H',ns ∪ S'),proceed α β)
   else
    ((H,S),termfalse α β (Γ,Δ))
 end.
```

Figure 3: The function `step`.

The `step` function proceeds as follows: it obtains a pair (Γ, Δ) from the working set S, generates new derivatives by a symbol

$$(\Lambda, \Theta) = (\partial_a(\Gamma), \partial_a(\Delta))$$

and adds to S all the (Λ, Θ) that are not elements of $\{(\Gamma, \Delta)\} \cup H$. This is implemented by `ReW_pdrv_set_filtered` which prevents the whole process from entering potential infinite loops since each derivative is considered only once during the execution of `equivP`.

The return type of `step` is

```
((set (ReW α β) * set (ReW α β)) * step_case)
```

where the first component corresponds to the pair (H,S), constructed as described above. The second component is a term of type `step_case` which has the purpose of guiding the iteractive process of computing the equivalence of the derivatives of α and β: if it is the term `proceed`, then the iterative process should continue; if it is a term `termtrue` H then the process should terminate and H contains the set of all the derivatives of α and β. Finally, if it is a term `termfalse` (Γ, Δ), then the process should terminate. The pair (Γ, Δ) is a witness that $\alpha \not\sim \beta$, since $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$.

3.2.3 Implementation and termination of `equivP`.

The formalisation of `equivP` in the `Coq` proof assistant is presented in Fig.5. Its main component is the function `iterate` which is responsible for the iterative process of calculating the derivatives of α and β, or to find a witness that $\alpha \not\sim \beta$ if that is the case. The function `iterate` executes recursively until the function `step` returns either a term `termtrue` H, or a term `termfalse` (Γ, Δ). Depending on the result of `step`, the function `iterate` returns a term of type `term_cases`, which can be the term `Ok` H indicating that $\alpha \sim \beta$, or the term `NotOk` (Γ, Δ) indicating that $\alpha \not\sim \beta$, respectively.

A peculiarity of the `Coq` proof assistant is that it only accepts *provably terminating functions*, *ie.*, it only accepts *structurally decreasing* functions. Nevertheless, *general recursive* functions can be expressed in `Coq` *via* an encoding into structural recursive functions. The `Function` [BC02] command helps users to define such functions which are not structurally decreasing along with an evidence of its termination, as an illustration of the *certified programming paradigm* that `Coq` promotes. In the case of `iterate` such evidence is given by the proof that its recursive calls follow a well-founded relation.

The decreasing measure (of the recursive calls) for `iterate` is defined as follows: in each recursive call the cardinal of the accumulator set H increases by one element due to the computation of `step`. This increase of H can occur only less than

$$2^{|\alpha|_\Sigma+1} \times 2^{|\beta|_\Sigma+1} + 1$$

times, due to the upper bounds of the cardinalities of $PD(\alpha)$ and of $PD(\beta)$. Therefore, in each recursive call of `iterate`, if

$$\texttt{step}\, H\ S\ \_ = (H', \_, \_)$$

then the following condition holds:

$$(2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - \#H' < (2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - \#H \tag{8}$$

The relation `LLim` presented in Fig.4 defines the decreasing measure imposed by equation (8). Furthermore, the definition of `iterate` requires an argument of type `DP` α β that imposes that the accumulator set H and the working set S are invariantly disjoint along the computation of `iterate` which is required to ensure that the set H is always increased by one element at each recursive call.

Besides the requirement of defining `LLim` to formalise `iterate`, we had to deal with two implementation details: first, we have used the type `N` which is a binary representation of natural numbers provided by `Coq`'s standard library [Thea], instead of the type `nat` so that the computation of `MAX` becomes feasible for large natural numbers. The second detail is related to the computation over terms representing well founded relations: instead of using the proof `LLim_wf` directly in `iterate`, we use the proof returned by the call to the function `guard` that lazily adds 2^n constructors `Acc_intro` in front of `LLim_wf` so that the actual proof is never reached in practice, while maintaining the same logical meaning. This technique avoids normalisation of well founded relation proofs which is usually highly complex and may take too much time.

Finally, the function `equivP` is defined as a call to `equivP_aux` with the correct input, *ie.*, with the accumulator set $H = \emptyset$ and with the working set $S = \{(\{\alpha\}, \{\beta\})\}$. The function `equivP_aux` is a wrapper that pattern matches over the term of type `term_cases` returned by `iterate` and returns the corresponding Boolean value.

3.2.4 Correctness and completeness.

To prove the correctness of `equivP` we must prove that, if `equivP` returns `true`, then `iterate` generates all the derivatives and prove that all these derivatives agree on the *ewp* of its

```
Definition lim_cardN (z:N) : relation (set A) :=
  fun x y:set A => nat_of_N z - (cardinal x) < nat_of_N z - (cardinal y).

Lemma lim_cardN_wf : ∀ z, well_founded (lim_cardN z).

Section WfIterate.
  Variables α β : re.

  Definition MAX_fst := |α|Σ + 1.
  Definition MAX_snd := |β|Σ + 1.

  Definition MAX := (2^MAX_fst × 2^MAX_snd) + 1.
  Definition LLim := lim_cardN (ReW α β) MAX.

  Theorem LLim_wf : well_founded LLim.

  Fixpoint guard (n : nat)(wfp : well_founded (LLim)) : well_founded (LLim):=
   match n with
   |O => wf
   |S m => fun x => Acc_intro x (fun y _ => guard m (guard m wfp) y)
   end.

End WfIterate.
```

Figure 4: The decreasing measure of `iterate`.

components. To prove that all derivatives are computed, it is enough to ensure that the `step` function returns a new accumulator set H' such that:

$$\texttt{step}\ H\ S\ sig = (H', S', \_) \ \rightarrow\ \forall (\Gamma, \Delta) \in H',\, \forall a \in \Sigma,\, \partial_a(\Gamma, \Delta) \in (H' \cup S') \tag{9}$$

The predicate `invP` and the lemma `invP_step` presented in Fig.6 prove this property. This means that, in each recursive call to `iterate`, the sets H and S hold all the derivatives of the elements in H. At some point of the execution, by the finiteness of the number of derivatives, H will contain all such derivatives and S will eventually become empty. Lemma `invP_iterate` proves this fact by a proof by functional induction over the structure of `iterate`. From lemma `invP_equivP` we can prove that

$$\forall w \in \Sigma^\star, (\partial_w(\alpha), \partial_w(\beta)) \in \texttt{equivP}\, \emptyset\, \{(\{\alpha\}, \{\beta\})\} \tag{10}$$

by induction over the word w and using the invariants presented above.

To finish the correctness proof of `equivP` one needs to make sure that all the derivatives (Γ, Δ) verify the condition $\varepsilon(\Gamma) = \varepsilon(\Delta)$. For that, we have defined the predicate `invP_final` which strengthens the predicate `invP` by imposing that the previous property is verified. The predicate `invP_final` is proved to be an invariant of `equivP` and this implies *re* equivalence by equation (6), as stated by theorem `invP_final_eq_lang`.

For the case of completeness, it is enough to reason by contradiction: assuming that $\alpha \sim \beta$ then it must be true that

$$\forall w \in \Sigma^\star,\, \varepsilon(\partial_w(\alpha)) = \varepsilon(\partial_w(\beta))$$

which implies that `iterate` may not return a set of pairs that contain a pair (Γ, Δ) such that $\varepsilon(\Gamma) \neq \varepsilon(\Delta)$ and so, `equivP` must always answer `true`.

Using the lemmas `equivP_correct` and `equivP_correct_dual` of Fig.6 a tactic was developed to prove automatically the (in)equivalence of any two *re*'s α and β. This tactic

```
Inductive term_cases α β : Type :=
|OK : set (ReW α β) → term_cases α β | NotOk : ReW α β → term_cases α β.

Inductive DP (α β:re)(H S: set (ReW α β)) : Prop :=
| is_dp : H ∩ S = ∅ → c_of_ReW_set α β H = true → DP α β H S.

Lemma DP_upd : ∀ (h s : set (ReW α β)) (sig : set A), DP α β h s →
  DP α β (fst (fst (step α β h s sig))) (snd (fst (step α β h s sig))).

Function iterate(α β:re)(H S:set (ReW α β))(sig:set A)(D:DP α β h s)
 {wf (LLim α β) H}: term_cases α β :=
  let ((H',S',next) := step H S in
   match next with
   |termfalse x => NotOk α β x
   |termtrue  h => Ok α β h
   |progress    => iterate α β H' S' sig (DP_upd α β H S sig D)
  end.
Proof.
 (* Proof that LLim is a decreasing measure for iterate *)
 exact(guard r1 r2 100 (LLim_wf r1 r2)).
Defined.

Definition equivP_aux(α β:re)(H S:set (ReW α β))(sig:set A)(D:DP α β H S):=
 let H' := iterate α β H S sig D in
  match H' with
  | Ok _     => true   | NotOk _ => false
 end.

Definition mkDP_ini : DP α β ∅ {ReW_1st α β} := (* ... *).

Definition equivP (α β:re)(sig:set A) :=
 equivP_aux α β ∅ {ReW_1st α β} sig (mkDP_ini α β).
```

Figure 5: Implementation of `equivP`

.

works by reducing the logical proof of the (in)equivalence of *re*'s into a Boolean equality involving the computation of `equivP`. After effectively computing `equivP` into a Boolean constant, the rest of the proof amounts at applying the reflexivity of Coq's primitive equality. Note that this tactic is also able to solve *re* containment due to the equivalence

$$\alpha \leq \beta \leftrightarrow \alpha + \beta \sim \beta \tag{11}$$

4 Concluding remarks and applications

In this paper we have described the formalisation of the procedure `equivP` for deciding *re* equivalence based in partial derivatives. This procedure has the advantage of not requiring the normalisation modulo ACI of *re*'s in order to prove its termination. Furthermore, the procedure `equivP` includes a refutation step that allows to prove the inequivalence of two *re*'s without the need to compute all their derivatives, which considerably improves its efficiency.

We have implemented `equivP` with no focus on its computational efficiency, but rather with the goal of providing a mechanically verified evidence that the algorithm suggested by Almeida *et al* [AMPMdS11] is correct. However, the performance exhibited by `equivP` is acceptable in the sense that it is able to prove "human written" equivalences (or *re* containment) almost instantaneously, and it is even faster when deciding *re*'s (in)equivalences due to the refutation step built-in in the procedure.

```
Definition invP (α β:re)(H S:set (ReW α β))(sig:set A) :=
 ∀ x, x \In H → ∀a, a \In sig → (ReW_pdrv α β x a) \In (H ∪ S).

Lemma invP_step : ∀ H S sig,
 invP H S sig → invP (fst (fst (step α β H S sig)))
                     (snd (fst (step α β H S sig))) sig.

Lemma invP_iterate : ∀ H S sig D,
 invP H S sig → invP (iterate α β H S sig D) ∅ sig.

Lemma invP_equivP :
 invP (equivP α β Σ) ∅ Σ.

Definition invP_final (α β:re)(H S:set (ReW α β))(sig:set A) :=
 (ReW_1st α β) \In (H ∪ S) /\
 (∀ x, x ∈ (H ∪ S) → c_of_ReW α β x = true) /\ invP α β H S sig.

Lemma invP_final_eq_lang :
 invP_final α β (equivP α β Σ) ∅ Σ → α ∼ β.

Theorem equivP_correct : ∀ α β, equivP α β sigma = true → α ∼ β.
Theorem equivP_complete : ∀ α β, α ∼ β → equivP α β sigma = true.
Theorem equivP_correct_dual : ∀ α β, equivP α β sigma = false → α ≁ β.
Theorem equivP_complete_dual : ∀ α β, α ≁ β → equivP α β sigma = false.
```

Figure 6: Invariants of `step` and `iterate`.

The purpose of this research is part of a broader project, where we plan to use Kleene algebra with tests to reason about the partial correctness of programs. The idea is to use the formalised decision procedure described in this paper as a certified trust-base to compare KAT terms once these terms are reduced into KA terms [Coh94, Wor08].

References

[AM94] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. In G. Rozenberg and A. Salomaa, editors, *Developments in Language Theory*, pages 195 – 209. World Scientific, 1994.

[AMPMdS11] José Bacelar Almeida, Nelma Moreira, David Pereira, and Simão Melo de Sousa. Partial derivative automata formalized in Coq. In M. Domaratzki and K. Salomaa, editors, *CIAA'10: Proceedings of the 15th international Conference on Implementation and Application of Automata*, number 6482 in LNCS, pages 59–68. Springer-Verlag, 2011.

[AMR08] Marco Almeida, Nelma Moreira, and Rogério Reis. Antimirov and Mosses's rewrite system revisited. In O. Ibarra and B. Ravikumar, editors, *CIAA 2008: 13th International Conference on Implementation and Application of Automata*, number 5448 in LNCS, pages 46–56. Springer-Verlag, 2008.

[AMR10] M. Almeida, N. Moreira, and R. Reis. Testing regular languages equivalence. *Journal of Automata, Languages and Combinatorics*, 15(1/2):7–25, 2010.

[BC02] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In Victor Carreño, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.

[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.

[BP10] Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.

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

[Coh94] Ernie Cohen. Hypotheses in Kleene algebra. Technical report, 1994.

[CS] Thierry Coquand and Vincent Siles. A decision procedure for regular expression equivalence in type theory. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP 2011, Kenting, Taiwan, December 7-9, 2011.*, number 7086 in LNCS, pages 119–134. Springer-Verlag.

[CZ01] Jean-Marc Champarnaud and Djelloul 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. Research Report 97–04, LIP - ENS Lyon, February 1997.

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

[HS07] P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfenning, editor, *CADE 2007*, number 4603 in LNAI, pages 279–294. Springer-Verlag, 2007.

[Kle] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, shannon, C. and McCarthy, J. edition.

[KN11] Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automated Reasoning*, 2011. Published online.

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

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

[McC] William McCune. Prover9 and Mace4. `http://www.cs.unm.edu/smccune/mace4`. Access date: 1.10.2011.

[Mir66] B.G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.

[MP08] Nelma Moreira and David Pereira. KAT and PHL in Coq. *Computer Science and Information Systems*, 05(02), December 2008. ISSN: 1820-0214.

[MPaM] Nelma Moreira, David Pereira, and Sim ao Melo de Sousa. Source code of the formalization. `http://www.liacc.up.pt/~kat/equivP.tgz`.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[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.

[Thea] The Coq Development Team. Coqlib. `http://coq.inria.fr/stdlib/`.

[Theb] The Ssreflect Development Team. Ssreflect. `http://www.msr-inria.inria.fr/Projects/math-components`.

[Wor08] James Worthington. Automatic proof generation in Kleene algebra. In *RelMiCS'08/AKA'08*, volume 4988 of *LNCS*, pages 382–396, Berlin, Heidelberg, 2008. Springer-Verlag.